Advances in Database Technology — EDBT 2021

24th International Conference
on Extending Database Technology
Nicosia, Cyprus, March 23–26, 2021
Proceedings

Editors

Yannis Velegrakis
Demetris Zeinalipour
Panos K. Chrysanthis
Francesco Guerra

open proceedings

Advances in Database Technology — EDBT 2021
Proceedings of the 24th International Conference
on Extending Database Technology
Nicosia, Cyprus, March 23–26, 2021

Series ISSN: 2367-2005

Editors

Yannis Velegrakis, University of Trento, Italy and Utrecht University, Netherlands
Demetris Zeinalipour, University of Cyprus, Cyprus
Panos K. Chrysanthis, University of Cyprus, Cyprus and University of Pittsburgh, USA
Francesco Guerra, University of Modena and Reggio Emilia, Italy

OpenProceedings.org
University of Konstanz
University Library
78457 Konstanz, Germany

OpenProceedings ISBN: 978-3-89318-084-4

DOI of this front matter: 10.5441/002/edbt.2021.01

Foreword by the PC Chair

The International Conference on Extending Database Technology (EDBT) is an established forum for researchers and practitioners alike to disseminate knowledge and research results related to data management. This year, the 24th edition of EDBT, scheduled to take place between March 23rd and March 26, 2021 in Nicosia, Cyprus, has instead been held entirely online, due to the circumstances and the travel restrictions imposed by the 2020/2021 pandemic. It has been jointly organized with the International Conference on Database Theory (ICDT).

The organizing committee solicited contributions in many different areas, including, but not limited to, Data Preparation, Data Privacy, Database Engines, Distributed Data Systems, Graph Management, Reasoning over Data, Machine Learning and AI in Databases, Novel Database Architectures, Semi-structured Data Management, and User Interfaces for Data. Novel in this year's edition is the solicitation of contributions in the area of Applied Database Systems for Data Science. The goal is to give the opportunity to researchers from different areas dealing with interesting data management challenges in the context of Data Science, to disseminate them to the data management community, and at the same time to give database researchers working on interesting data science scenarios to publish their works.

The main Research track, the Industrial & Application track, as well as the Demonstration track have remained as in previous years, while the Short paper track has been slightly extended to accommodate papers of 6 pages, to allow more technical content.

With some rare exceptions, research papers were reviewed by 4 reviewers. Discussions and decisions were taken under the coordination of a senior PC member, with expertise in the respective area of the paper under review. The reviewing phase involved two cycles: resulting in a number of papers being revised and having their qualify improved significantly.

Consistent with the tradition, the program included two keynotes, one on Data Profiling by Felix Naumann (Hasso Plattner Institute, Germany), and one on Knowledge Management by Katja Hose (Aalborg University, Denmark). These keynotes were complemented by the two additional keynotes of the co-organized ICDT. Last but not least, the program had the traditional tutorial track, featuring four tutorials on topics related to knowledge graphs, blockchains, text-to-SQL, and time series management. The overall program was accompanied by 6 Workshops.

The Program committee reviewed 108 full research papers, of which 27 were accepted. For short papers, 34 papers out of 113 were selected. The Industry and Application track received 22 submissions of which it selected 11 for publication, while the demo track received 24 works of which 15 were accepted.

Given that this year the conference took place online, it was a good opportunity to exploit at the maximum the opportunities that digital technologies can offer. Furthermore, it was our intention to make sure that the paper contributions are disseminated to a broader audience, especially outside the conference participants. For this reason, the authors were asked to provide a pre-recorded presentation of 10 min that will remain in the proceedings, a 30 sec pitch video that advertises the results of the contribution and a graphics ad.

In order to recognize significant contributions and give credits to the authors, the technical program committee awarded the Best Paper award to one of the research papers and the Best Demonstration award to one of the demos. Furthermore, following the EDBT tradition, it awarded the Test-of-Time award to a paper from the EDBT 2011 proceedings that has been deemed to have the greatest impact among those of that year. A novelty in this year's EDBT edition is the additional recognition of the Best Short Paper.

The realization of EDBT 2021 is a result of a collaborative effort of the different chairs and program committee members. Congratulations are in place to the senior PC members for guiding the discussions in such a professional and timely manner, ensuring the selection of the best quality papers. Special thanks for the excellent collaboration and quality of work go to the Industrial and Application Chair Eric Simon, the Demonstration Chair Sihem Amer-Yahia, the tutorial chairs Stefan Manegold and Wang-Chiew Tan, and the Workshop Chair Evaggelia Pitoura. A great deal of credits go to the Proceedings Chair Francesco Guerra for all the extra work he has put in organizing the material and making sure that all the proceedings information is in place. My appreciation goes also to the members of the paper award committees for the effort they put in evaluating the candidate papers and providing the decisions in a timely manner. I find it impossible to describe the passion, professionalism, consistency and collaborative attitude the general chairs Demetris Zeinalipour and Panos K. Chrysanthis have demonstrated. It was great working with them. I would also like to thank all the program committee members since due to them the high quality program became possible, the award committees, as well as Angela Bonifati and Marc H. Scholl from the EDBT Board for the their numerous advices and support. Last but not least, I would like to thank all the authors for the works they submitted to the conference, the keynote speakers, the tutorial presenters, and the demonstration presenters.

Yannis Velegrakis, EDBT 2021 PC Chair

Message from the General Chairs

The 24th edition of the International Conference on Extending Database Technology (EDBT) was held between the 23rd and 26th of March 2021, despite the COVID-19 worldwide pandemic. Continuing its longstanding tradition as a premier data management research forum taking place in a European location, EDBT 2021, jointly organized with the International Conference on Database Theory (ICDT), was held virtually in Nicosia, the capital of the beautiful island of Cyprus.

At the onset, our aim has been to offer as close to an in-person, face-to-face experience as possible, maximizing the dissemination of knowledge and sharing of research results among the EDBT/ICDT participants at no health risk to them, and together discover the Cypriot culture and hospitality. EDBT/ICDT 2021 was initially planned to become the first hybrid EDBT/ICDT conference, combining physical presentations with an extended online audience, although this plan later evolved into an online format due to the ongoing situation with COVID-19. To this end, we have been committed to offering the best possible online experience to attendees by capitalizing and expanding on the success of earlier conferences.

Particularly, we decided on a number of novelties compared to previous conferences. Firstly, given that an optimal online experience cannot be achieved with pre-recorded presentations, but only with live presentations and direct interactions between the speakers and the audience, we decided in coordination with the Program Chairs on the online synchronous delivery of all events, including all research presentations, tutorials and live demonstrations. The next challenge was to select the appropriate conference management platform. Our personal experience as participants in several online conferences during the past year, and our evaluation of a number of popular web socializing platforms, led us to the decision that a custom conference attendance experience can only be delivered by an in-house platform that relies on a reliable teleconferencing channel, an intuitive interface, and a low learning curve.

To this end, the Data Management Systems Laboratory at the University of Cyprus, under the leadership of Demetris Zeinalipour developed and hosted a novel web-based platform named VGATE (Virtual Gate) to online conferences. VGATE allowed the Organizing Committee to collaborate over a Web-based Sheet interface (like Google Sheets) to plan and manage the organization. VGATE provided numerous helpful building-blocks in the online organization, namely, zoom license management and user status integration, integration of proceedings and multimedia content, industry booths, live sessions, social rooms, and for the first time an online helpdesk supported by Easy-Conferences Ltd. Special thanks to Paschalis Mpeis (University of Cyprus, Cyprus), Soteris Constantinou (University of Cyprus, Cyprus) and Constantinos Costa (University of Pittsburgh, USA) for their support and code contributions to the project under an extremely tight schedule. Special thanks also to Zoom Video Communications, Inc., for facilitating and sponsoring EDBT/ICDT 2021.

Novel this year is the participation of EDBT/ICDT to the newly founded Diversity and Inclusion (D&I) initiative of the Data Management community. EDBT/ICDT (alongside SIGMOD, VLDB, SoCC, and ICDE) celebrates the diversity in our community and welcomes everyone regardless of age, sex, gender identity, race, ethnicity, socioeconomic background, country of origin, religion, sexual orientation, physical ability, education, work experience, etc. To introduce this initiative, Panos K. Chrysanthis, the EDBT/ICDT D&I Chair, together with Sihem Amer-Yahia (CNRS, University Grenoble Alpes, France), a D&I Core Member, organized a panel as part of the Reception on DAY1 of the conference.

With D&I and broad participation in mind, the conference program was structured in a way that is convenient to at least half of the world at any given time. Specifically, the program was split into morning sessions (CET), which were convenient for EU-ASIA participants, on DAY1 and DAY3, and the afternoon sessions (CET), which were convenient for EU-America participants on DAY2 and DAY4. The Workshops were scheduled on DAY1 in the morning or afternoon based on the geographical location of their presenters. Social events were split along the same lines, presenting the host country, Cyprus, through music and videos on culture, geography, food, leisure, and other enjoyable aspects of Cyprus. Virtual corridor and hallway discussions were supported by a number of unmoderated and dynamic social rooms on VGATE at all times.

This year we also introduced some additional novelties, namely: (i) We introduced a special ceremony during Reception on DAY2, titled "*Pandemic Greetings from the Pioneers and the Next Data Management Challenge*", with greetings by Philip A. Bernstein (Microsoft Research, WA, USA), Laura M. Haas (University of Massachusetts – Amherst, MA, USA), Yannis Ioannidis (University of Athens, Greece) and Jeffrey D. Ullman (Stanford, CA, USA); (ii) we introduced the concept of a sponsored industry talk during Dinner on DAY3, with title "*Behind the Scenes of Snowflake's new Search Optimization Service*", by Ismail Oukid (Snowflake, Germany) and Stefan Richter (Snowflake, Germany); (iii) we introduced a dedicated "*Demos in Action: Meet the Authors!*" session during Dinner on DAY3, which allowed Demo presenters to individually showcase their demos to participants in their private presentation space through

VGATE. Finally, we also continued to support the Climate Change discussion with a session on DAY4, led by Antoine Amarilli (Télécom Paris, France), with guest Benjamin Pierce (University of Pennsylvania, USA).

In all above endeavors, we had the support and encouragement of our incredible Organization Committee and the EDBT Executive Board, in particular, the President of EDBT Executive Board Angela Bonifati (Lyon 1 University, France), as well as the Chair of the ICDT Council Wim Martens (University of Bayreuth, Germany).

We are grateful to the entire Technical Program Committees, which, under the excellent leadership of the EDBT Program Chair Yannis Velegrakis (University of Trento, Italy and Utrecht University, Netherlands) and the ICDT Program Chair Ke Yi (Hong Kong University of Science and Technology, Hong Kong), brought forward an exciting technical program with a rich production of technical content (proceedings, videos, pitches, ads, etc.). Our gratitude also extends personally to the EDBT Demonstration Chair Sihem Amer-Yahia (CNRS, University Grenoble Alpes, France), the EDBT Workshop Chair Evaggelia Pitoura (University of Ioannina, Greece), the Climate Chair Antoine Amarilli (Télécom Paris, France), the EDBT Applied Database Systems for Data Science Vice-Chair Paul Groth (University of Amsterdam, Netherlands), the EDBT Industrial/Application Chair Eric Simon (SAP, France), the Tutorial Chairs Stefan Manegold (CWI, Netherlands) and Wang-Chiew Tan (Megagon Labs, USA), for the excellent collaboration, and exchange of ideas and insightful discussions. We would also like to highlight EDBT Demonstration Chair Sihem Amer-Yahia's success in putting together EDBT's first all-women Demonstration Track Committee. We would also like to thank the organizers of the six, co-located workshops DOLAP, BigVis, BMDA, DARLI-AP, SIMPLIFY and PIE+Q for enriching the scope of the technical program.

Several people contributed to the successful organization of the EDBT/ICDT 2021 conference. Special thanks to the following valued collaborators for their passion in organizing a memorable event: the Sponsorship Chair Divyakant Agrawal (University of California-Santa Barbara, USA), the Publicity Chair Herodotos Herodotou (Cyprus University of Technology, Cyprus), the Finance Chair George Pallis (University of Cyprus, Cyprus), the EDBT Proceedings Chair Francesco Guerra (University of Modena and Reggio Emilia, Italy), the ICDT Proceedings Chair Zhewei Wei (Renmin University, China), the Workshops Proceedings Chair Constantinos Costa (University of Pittsburgh, USA) and the Website support by Nicolas Kantzilaris (Easy Conferences, Cyprus).

Our sincere gratitude to our platinum sponsor, Snowflake, and our bronze sponsors, Oracle and Zoom, as well as the contact persons behind the support, namely Martin Hentschel (Snowflake), Ann Brisson (Oracle) and Alberto Colautti (Zoom).

Organizing EDBT 2021 at the University of Cyprus was Prof. George Samaras†' passion. After his unexpected passing two years ago, his friends and colleagues at the University of Cyprus volunteered to make his passion a reality. We are grateful to the EDBT Executive Board for giving us this opportunity and trusting us to organize EDBT 2021 alongside ICDT 2021 in Cyprus, as proposed by George in 2018. We dedicate the EDBT/ICDT 2021 in honor of the memory of Prof. George Samaras† (1959–2018).

We hope you enjoyed EDBT/ICDT 2021's exciting technical program and your virtual visit to Cyprus!

Demetris Zeinalipour, University of Cyprus, Cyprus
Panos K. Chrysanthis, University of Cyprus, Cyprus and University of Pittsburgh, USA

EDBT/ICDT 2021 General Co-Chairs

Program Committee Members

Research Program Committee Chair

Yannis Velegrakis, U Trento, Italy & Utrecht U, The Netherlands

Senior Program Committee Members

Periklis Andritsos, U Toronto, Canada
Nikos Bikakis, Athena Res. Ctr., Greece
Elena Ferrari, U Insubria, Italy
Avigdor Gal, Technion – Israel IT, Israel
Lukasz Golaz, U Waterloo, Canada
Paul Groth, U Amsterdam, Netherlands
Sergio Greco, U Calabria, Italy
Christian S. Jensen, Aalborg U, Denmark
Laks V.S. Lakshmanan, U British Columbia, Canada
Qiong Luo, HKUST, China
Raymond Ng, U British Columbia, Canada
Tamer Özsu, U Waterloo, Canada
Pinar Tozun, IT U Copenhagen, Denmark

Program Committee Members

Karl Aberer, EPFL, Switzerland
Ashraf Aboulnaga, Qatar Comp. Res. Inst., HBKU, Qatar
Bernd Amann, Sorbonne U, France
Nicolas Anciaux, INRIA, France
Walid G. Aref, Purdue U, USA
Akhil Arora, EPFL, Switzerland
Manos Athanassoulis, Boston U, USA
Nikolaus Augsten, U Salzburg, Austria
Elena Baralis, Politecnico di Torino, Italy
Denilson Barbosa, U Alberta, Canada
Ilaria Bartolini, U Bologna, Italy
Senjuti Basu Roy, New Jersey IT, USA
Luigi Bellomarini, Banca d'Italia, Italy
Michael Benedikt, U Oxford, UK
Alexander Boehm, SAP SE, Germany
Luc Bouganim, INRIA, France
Andrea Cali, Birkbeck U London, UK
Marco Calautti, U Trento, Italy
Bogdan Cautis, U Paris-Sud, France
Mel Chekol, Utrecht U, Netherlands
Vassilis Christophides, ENSEA, ETIS, France
Dario Colazzo, U Paris Dauphine – PSL, France
Bin Cui, Peking U, China
Alfredo Cuzzocrea, ICAR-CNR & U Calabria, Italy
Sabrina De Capitani di Vimercati, U Milano, Italy
Antonios Deligiannakis, TU Crete, Greece
Çağatay Demiralp, Sigma Computing, USA
Stefania Dumbrava, ENSIIE, France
George Fakas, Uppsala U, Sweden
Ju Fan, Renmin U China, China
Donatella Firmani, Roma Tre U, Italy
George Fletcher, Eindhoven UT, Netherlands
Daniele Foroni, Huawei, Germany
Johann-Christoph Freytag, HU Berlin, Germany
Avigdor Gal, Technion — Israel IT, Israel
Johann Gamper, Free U Bozen-Bolzano, Italy
Yunjun Gao, Zhejiang U, China
Rainer Gemulla, U Mannheim, Germany
Boris Glavic, Illinois IT, USA
Paolo Guagliardo, U Edinburgh, UK
Michael Gubanov, Florida State U, USA
Xi He, U Waterloo, Canada
Melanie Herschel, U Stuttgart, Germany
Jan Hidders, U London, UK
Katja Hose, Aalborg U, Denmark
Vagelis Hristidis, U California – Riverside, USA
Haoyu Huang, Google, USA
Xin Huang, Hong Kong Baptist U, Hong Kong SAR
Ekaterini Ioanou, Tilburg U, Netherlands
Zsolt István, ITU Copenhagen, Denmark
Panos Kalnis, King Abdullah UST, Saudi Arabia
Vana Kalogeraki, Athens U Eco. & Busin., Greece
Verena Kantere, National TU Athens, Greece
Panagiotis Karras, Aarhus U, Denmark
Asterios Katsifodimos, TU Delft, Netherlands
Anastasios Kementsietsidis, Google Research, USA
Haridimos Kondylakis, FORTH-ICS, Greece
Nick Koudas, U Toronto, Canada
Georgia Koutrika, Athena Research Center, Greece
Alexandros Labrinidis, U Pittsburgh, USA
Ulf Leser, HU Berlin, Germany
Guoliang Li, Tsinghua U, China
Han Li, Amazon, USA
Xiang Lian, Kent State U, USA
Chunbin Lin, Amazon AWS, USA
Matteo Lissandrini, Aalborg U, Denmark
Eric Lo, Chinese U Hong Kong, Hong Kong SAR
Ping Lu, Beihang U, China
Nikos Mamoulis, U Ioannina, Greece
Ioana Manolescu, INRIA & Inst. Poly. Paris, France
Sebastian Michel, TU Kaiserslautern, Germany
Paolo Missier, Newcastle U, UK

Mohamed Mokbel, U Minnesota – Twin Cities, USA
Mirella M. Moro, U Federal Minas Gerais, Brazil
Davide Mottin, Aarhus U, Denmark
Hieu Nguyen, eBay, USA
Behrooz Omidvar-Tehrani, LIG, France
Mourad Ouzzani, Qatar Comp. Res. Inst., HBKU, Qatar
George Papadakis, U Athens, Greece
Olga Papaemmanouil, Brandeis U, USA
Odysseas Papapetrou, TU Eindhoven, Netherlands
Paolo Papotti, Eurecom, France
Alok Pareek, Striim, USA
Torben Bach Pedersen, Aalborg U, Denmark
Eric Peukert, Leipzig U, Germany
Dimitris Plexousakis, ICS-FORTH, Greece
Laura Po, U Modena & Reggio Emilia, Italy
Arnau Prat, Sparsity Technologies, Spain
Nicoleta Preda, U Versailles, France
Abdulhakim Qahtan, Utrecht U, Netherlands
Louiqa Raschid, U Maryland, USA
Mohammad Sadoghi, U California, Davis, USA
Carlo Sartiani, U Basilicata, Italy
Kai-Uwe Sattler, TU Ilmenau, Germany
Sebastian Schelter, U Amsterdam, Netherlands
Steffi Scherzinger, U Passau, Germany
Petra Selmer, Neo4j, UK
Juan Sequeda, data.world, USA
Lidan Shou, Zhejiang U, China
Giovanni Simonini, U Modena & Reggio Emilia, Italy
Hala Skaf-Molli, U Nantes, France
Kostas Stefanidis, Tampere U, Finland
Gabor Szarnyas, CWI, Netherlands
Letizia Tanca, Politecnico di Milano, Italy
Ernest Teniente, U Politècnica Catalunya, Spain
Arash Termehchy, Oregon State U, USA
Chao Tian, Alibaba, China
Riccardo Torlone, Roma Tre U, Italy
Farouk Toumani, Clermont Auvergne U, CNRS, France
Katerina Tzompanaki, CY Cergy Paris U, France
Vasilis Vassalos, Athens U Eco. & Busin., Greece
Panos Vassiliadis, U Ioannina, Greece
Yannis Velegrakis, Utrecht U, Netherlands
Jianguo Wang, Purdue U, USA
Wendy Hui Wang, Stevens IT, USA
Wolfram Wingerath, Baqend, Germany
Nikolay Yakovets, TU Eindhoven, Netherlands
Meihui Zhang, Beijing IT, China

Industrial Track Program Committee

Tyler Akidau, Snowflake, USA
Minhea Andrei, SAP, France
Angela Bonifati, U Lille, France
Jianjun Chen, Bytedance US Lab, USA
Thomas Fanghaenel, Salesforce, USA
Avrilia Floratou, Microsoft, USA
Prasanta Ghosh, Microsoft, USA
Yash Govind, Informatica LLC, USA
Laura Haas, U Mass. Amherst, USA
Zack Ives, U Pennsylvania, USA
Martin Kersten, MonetDB Solutions, Netherlands
Hariharan Lakshmanan, Oracle, USA
Dustin Lange, Amazon Research, USA
Jyoti Leeka, Microsoft, USA
Stefan Mandl, EXASOL AG, Germany
Anisoara Nica, SAP Labs Waterloo, Canada
Berthold Reinwald, IBM Research-Almaden, USA
Alejandro Salinger, SAP SE, Germany
Dennis Shasha, New York U, USA
Mohamed Soliman, Datometry, USA
Nesime Tatbul, Intel Labs and MIT, USA
Wei Wang, Nat. U Singapore, Singapore
Xuezhi Wang, Google, USA
Yongsik Yoon, Snowflake, USA
Kai Zeng, Alibaba Group, China

Demonstration Track Program Committee

Anastasia Ailamaki, EPFL, Switzerland
Elena Baralis, Polytecnico di Torino, Italy
Senjuti Basu Roy, NJIT, USA
Angela Bonifati, Lyon U, France
Renata Borovica-Gajic, U Melbourne, Australia
Malu Castellanos, TERADATA, USA
Sarah Cohen Boulakia, U Paris Sud, France
Maria Luisa Damiani, U Milan, Italy
Anna Fariha, UMass Amherst, USA
Irini Fundulaki, FORTH, GRnet, Greece
Katja Hose, Aalborg U, Denmark
Vana Kalogeraki, Athens U Eco. & Busin., Greece
Zoi Kaoudi, TU Berlin, Germany
Georgia Koutrika, ATHENA, Greece
Ioana Manolescu, INRIA, France
Renée J. Miller, Northeastern U, USA
Silvia Nittel, U Maine, USA
Fatma Özcan, Google, USA
Nesime Tatbul, Intel Labs & MIT, USA
Pinar Tözün, ITU, Denmark
Esther Pacitti, U Montpellier (INRIA & CNRS), France
Danica Porobic, Oracle, Switzerland
Agma Traina, ICMC-USP, Brazil
Zografoula Vagena, U Paris, France and RelationalAI, USA
Xiaolan Wang, Megagon Labs, USA
Karine Zeitouni, U Versailles, France

Additional Reviewers

Yaniv Gur, IBM Research-Almaden, USA
Ahmed Al-Baghdadi, Kent State U, USA
Niranjan Rai, Kent State U, USA
Weilong Ren, Kent State U, USA
Tahereh Arabghalizi, U Pittsburgh, USA
Evangelos Karageorgos, U Pittsburgh, USA
Xiaoting Li, U Pittsburgh, USA
Anthony Sicilia, U Pittsburgh, USA
Prithu Banerjee, U BC, Vancouver, Canada
Saket Gurukar, Ohio State U, Ohio, USA
Garima Gaur, IIT Kanpur, India
Sainyam Galhotra, U Mass., Amherst, USA
Nikos Giatrakos, TU Crete, Greece
Angjela Davitkova, TU Kaiserslautern, Germany
Leonardo Gazzarri, U Stuttgart, Germany
Flavio Giobergia, Politecnico di Torino, Italy
Eliana Pastor, Politecnico di Torino, Italy
Uta Störl, Darmstadt U of Applied Sciences, Germany
Wolfgang Mauerer, TU of Appl. Sc. Regensburg, Germany
Vasilis Efthymiou FORTH ICS Greece
Nikos Myrtakis U Crete, Greece
Benjamin Wollmer, Baqend and U Hamburg, Germany
Andrea Rossi, U Roma Tre, Italy
Tobias Lindaaker, Neo4j, Sweden

Conference Organization

General Chairs
Demetris Zeinalipour, University of Cyprus, Cyprus
Panos K. Chrysanthis, University of Cyprus, Cyprus and University of Pittsburgh, USA

EDBT Program Chair
Yannis Velegrakis, University of Trento, Italy and Utrecht University, Netherlands

ICDT Program Chair
Ke Yi, Hong Kong University of Science and Technology, Hong Kong

EDBT Industrial/Application Chair
Eric Simon, SAP, France

EDBT Applied Database Systems for Data Science Vice-Chair
Paul Groth, University of Amsterdam, Netherlands

EDBT Demonstrations Chair
Sihem Amer-Yahia, CNRS, Univ. Grenoble Alpes, France

Tutorial Chairs
Stefan Manegold, CWI, Netherlands
Wang-Chiew Tan, Megagon Labs, USA

Workshops Chair
Evaggelia Pitoura, University of Ioannina, Greece

Climate Chair
Antoine Amarilli, Télécom Paris, France

EDBT Proceedings Chair
Francesco Guerra, University of Modena and Reggio Emilia, Italy

ICDT Proceedings Chair
Zhewei Wei, Renmin University, China

Workshops Proceedings Chair
Constantinos Costa, University of Pittsburgh, USA

Diversity and Inclusion Chair
Panos K. Chrysanthis, University of Pittsburgh, USA

Sponsorship Chair
Divyakant Agrawal, UC Santa Barbara, USA

Publicity Chair
Herodotos Herodotou, Cyprus University of Technology, Cyprus

Finance Chair
George Pallis, University of Cyprus, Cyprus

Website Chair
Kyriakos Georgiades, EasyConferences, Cyprus

Test-of-Time Award

Established in 2014, the Test-of-Time Award of the Extended Database Technology (EDBT) Conference recognizes papers presented at the EDBT Conferences that have had the most impact in terms of research, methodology, conceptual contribution, or transfer to practice over the past ten years. The 2021 Test-of-Time Award committee looked and evaluated the impact of the papers in the EDBT 2011 proceedings, and selected

SeMiTri: a framework for semantic annotation of heterogeneous trajectories

by Zhixian Yan, Dipanjan Chakraborty, Christine Parent, Stefano Spaccapietra, and Karl Aberer

published in the EDBT 2011 Proceedings, pp. 259–270, DOI: 10.1145/1951365.1951398,

because it is one of the earliest papers to propose a general method for enriching moving object trajectories with semantics useful for supporting location-based services, which have been and still are in high demand across several application sectors. Since its publication, SeMiTri has generated significant interest, and follow-up work on semantic processing of mobile data and trajectories.

The EDBT 2021 Test of Time Award committee was formed by Barbara Catania, University of Genova, Italy, Gautam Das, University of Texas at Arlington, USA, Beng Chin OOI, National University of Singapore, Singapore, Themis Palpanas, University of Paris, France, and Yufei Tao, Chinese University of Hong Kong, China.

The EDBT Test-of-Time award for 2021 will be presented during the EDBT/ICDT 2021 Conference in Nicosia, Cyprus, as part of the Awards session on March 24, 2021.

Best Paper Award

The Best Paper Award Committee has looked at the papers accepted in the conference and selected one that was distinguishing itself in terms of research quality, presentation, technical challenges, and novelty. The selected paper is

DomainNet: Homograph Detection for Data Lake Disambiguation

by Aristotelis Leventidis, Laura Di Rocco, Wolfgang Gatterbauer,
Renée J. Miller and Mirek Riedewald.
DOI: 10.5441/002/edbt.2021.03

The paper presents DomainNet, a system that disambiguates values from heterogeneous datasets by creating a network representing co-occurring values and computing their graph centrality. The system is unsupervised, its accuracy outperforms the state-of-the-art, and it is accompanied by an open benchmark. The paper is of high significance: the problem is important, the proposed solution is effective, and the benchmark facilitates further research.

Abstract: Modern data lakes are deeply heterogeneous in the vocabulary that is used to describe data. We study a problem of disambiguation in data lakes: how can we determine if a data value occurring more than once in the lake has different meanings and is therefore a homograph? While word and entity disambiguation have been well studied in computational linguistics, data management and data science, we show that data lakes provide a new opportunity for disambiguation of data values since they represent a massive network of interconnected values. We investigate to what extent this network can be used to disambiguate values.
DomainNet uses network-centrality measures on a bipartite graph whose nodes represent values and attributes to determine, without supervision, if a value is a homograph. A thorough experimental evaluation demonstrates that state-of-the-art techniques in domain discovery cannot be re-purposed to compete with our method. Specifically, using a domain discovery method to identify homographs has a precision and a recall of 38% versus 69% with our method on a synthetic benchmark. By applying a network-centrality measure to our graph representation, DomainNet achieves a good separation between homographs and data values with a unique meaning. On a real data lake our top- 200 precision is 89%.

The EDBT 2021 Best Paper Award committee was formed by Avigdor Gal, Technion Israel Institute of Technology, Israel, Lucasz Golab, University of Waterloo, Canada, Christian Jensen, Aalborg University, Denmark, and Qiong Luo, HKUST, China.

The EDBT Best Paper Award for 2021 will be presented during the EDBT/ICDT 2021 Conference in Nicosia, Cyprus, on March 24, 2021.

Best Short Paper Award

The Best Short Paper Award Committee has looked at the papers accepted in the conference and selected one that was distinguishing itself in terms of research quality, presentation, technical challenges, novelty, and potential impact to the broader research community and industry. The selected paper is

Answer Graph: Factorization Matters in Large Graphs

by Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Stanley Clark, and Mark Chignell.
DOI: 10.5441/002/edbt.2021.56

The paper proposes a two-step method for answering SPARQL conjunctive queries. The first step constructs an answer graph consisting only of node-edge-node triples that are answers to the query (a factorized answer set). In the second step, this answer graph is used to compute the actual embeddings for the query. The authors present detailed implementations for generating answer graphs and computing the final embedding, particularly in the case of cyclic conjunctive queries. Some first results from an experimental evaluation are also provided.

The EDBT 2021 Best Short Paper Award committee was formed by Manos Athanasoulis, Boston University, USA, Johann Gamper, Free University of Bozen-Bolzano, Italy, Ioana Manolescu, INRIA, France, Letizia Tanca, University of Milan, Italy, and Yannis Kotidis, Athens University of Economics and Business, Greece

The EDBT Best Short Paper Award for 2021 will be presented during the EDBT/ICDT 2021 Conference in Nicosia, Cyprus.

Best Demonstration Award

The Best Demonstration Award Committee has reviewed the video recordings of the 15 demos accepted at EDBT/ICDT and selected the most engaging demonstration that serves as an example of a structured and well illustrated demonstration and showcases an end-to-end system on a state-of-the-art topic that promotes data management beyond its boundaries. The selected demo is

Conversational OLAP in Action

by Matteo Francia, Enrico Gallinucci, and Matteo Golfarelli
(DISI – University of Bologna)
DOI: 10.5441/002/edbt.2021.74

For demonstrating COOL, a tool supporting natural language COnversational OLap sessions. COOL interprets and translates a natural language dialogue into an OLAP session that starts with a GPSJ query. The demonstration is engaging and showcases the usability of COOL and its capabilities in assisting query formulation and ambiguity resolution.

The EDBT 2021 Best Demonstration Award committee was formed by Sihem Amer-Yahia, CNRS Univ. Grenoble Alpes, France, Elena Baralis, Politecnico di Torino, Italy, Maria Luisa Damiani, University of Milan, Italy, Anna Fariha, UMass Amherst, USA, Irini Fundulaki, FORTH, GRnet, Greece, Zoi Kaoudi, TU Berlin, Germany, Georgia Koutrika, ATHENA, Greece, Esther Pacitti, University of Montpellier (Inria&CNRS), France, and Agma Traina, ICMC-USP, Brazil.

The EDBT Best Demonstration Award for 2021 will be presented during the EDBT/ICDT 2021 Conference in Nicosia, Cyprus, on March 24, 2021.

Table of Contents

Industrial Papers

Demos

Exchanging Data under Policy Views

Angela Bonifati
Lyon 1 University & Liris CNRS
Lyon, France
angela.bonifati@univ-lyon1.fr

Ugo Comignani
Grenoble INP Ensimag & LIG CNRS
Grenoble, France
ugo.comignani@grenoble-inp.fr

Efthymia Tsamoura
Samsung AI Research
Cambridge, UK
efi.tsamoura@samsung.com

ABSTRACT

Exchanging data between data sources is a fundamental problem in many data science and data integration tasks. In this paper, we focus on the data exchange problem in the presence of privacy constraints on the source data, which has been disregarded in the literature to date. By leveraging a logical privacy-preservation paradigm, the privacy restrictions are expressed as a set of *policy views* representing the information that is safe to expose over *all* instances of the source in order to exchange them with the target. We introduce a protocol that provides formal privacy guarantees and is *data-independent*, i.e., under certain criteria, it guarantees that the mappings leak no sensitive information independently of the instances lying in the source. Moreover, we design an algorithm for *repairing* an input mapping w.r.t. a set of policy views, in cases where the input mapping leaks sensitive information. We show that the repairing can build upon hard-coded and learning-based user preference functions and we show the trade-offs. Our empirical evaluation shows that repairing mappings is quite efficient, leading to repairing sets of 300 s-t tgds in an average time of 5s on a commodity machine. It also shows that the repairing based on learning is robust and has comparable runtimes with the hard-coded one.

KEYWORDS

privacy-preserving data integration, data exchange, mapping repairs

1 INTRODUCTION

Data exchange is a key process in data science and data integration pipelines, leading to translating data compliant with a source schema S and lying in a source database to a target database with a non-overlapping target schema T [1, 4, 17]. Data exchange is also part of metadata management operations [6], since the schema mappings between source and target also known as *source-to-target* (s-t) dependencies Σ_{st} (s-t tgds) are declarative expressions manipulating schema elements, i.e. metadata rather than data.

Despite a wealth of research on the topic, the privacy-aware variant of the data exchange problem has received little attention to date. However, recent data protection regulations such as EU GDPR or CCPA in the US bring the attention to the problem of protecting personal data when transferring data across countries and institutions, thus motivating our work. In a privacy-aware data exchange scenario (as exemplified in Figure 1), the source schema comes with a set of constraints called policy views $\mathcal{V}$ representing the data that is *safe* to expose to the target over *all instances* of the source. The policy views can be considered as user views on the data of the source and can encode possible formulations of the different purposes the data will undergo during the exchange process as in many data protection regulations.

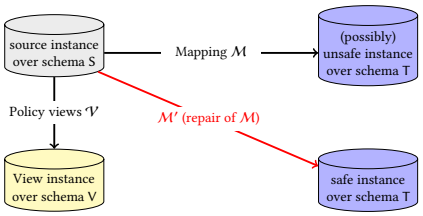


Figure 1: A privacy-aware data exchange setting with mappings and policy views.

This process entails the repairing of the original mapping $\mathcal{M}$ into a mapping $\mathcal{M}'$ in order to make the exported target instance safe. However, in order to realize such a data exchange scenario, one needs to address the following issues: (1) given a set of privacy restrictions on the source schema, what would it mean for a data exchange setting to be safe under the proposed privacy restrictions?; (2) assuming that the privacy-preservation protocol is fixed, how could we assess the safety of a data exchange setting w.r.t. the privacy restrictions and provide *strong guarantees* of no privacy leakage?; finally, in case of privacy violations, (3) how could we *repair* the s-t tgds (and transform the mapping $\mathcal{M}$ into a repaired mapping $\mathcal{M}'$)?

To address the first issue, we build upon prior work on the logical foundations of privacy-preserving data integration [5, 21], and we tailor them to a data exchange setting. Hence, we define a set of s-t tgds to be safe w.r.t. the policy views if *every positive information* that is kept secret by the policy views is also kept secret by the s-t tgds. As we will see in subsequent sections and contrarily to previous work, our proposed privacy-preservation protocol is *data-independent* allowing us to provide strong privacy-preservation guarantees over all instances of the sources. As such, our work leads to the first practical framework establishing privacy-conscious data exchange. The above addresses the second aforementioned issue in that it enables a schema-level enforcement of the privacy-preserving protocol with strong guarantees. Regarding the third issue, we propose a repairing algorithm for the proposed privacy-preservation protocol in case of detected unsafety. Since multiple repairs are possible, such an algorithm might leverage techniques for learning the user preferences during the repairing process, which is also a desirable feature in privacy enforcement over sensitive data. In order to further illustrate the relevance of our problem, we illustrate a running example inspired by a real-life data exchange process between two different hospitals in the UK[1].

1.1 Illustrative example

Consider the source schema S of NHS consisting of the following relations: P, H_N, H_S, O and S as illustrated in Figure 2 (a). Relation P stores for each person registered with the hospital, his insurance number, his name, his ethnicity group and his county. Relations H_N and H_S store for each patient who has been admitted to some hospital in the north or the south of UK, his

[1] https://www.nhs.uk/

Source schema $S = \{P(i, n, e, c); H_N(i, d); H_S(i, d); O(i, t, p); S(i, n, e, c)\}$

Target schema $T = \{EthDis(e, d); CountyDis(c, d); SO(e)\}$

View schemas $V = \{V_1(e, d); V_2(c, d); V_3(t, p); V_4(e)\}$

(a) Schemas S, T and V

$$P(i, n, e, c) \wedge H_N(i, d) \rightarrow V_1(e, d) \quad (1)$$

$$P(i, n, e, c) \wedge H_S(i, d) \rightarrow V_2(c, d) \quad (2)$$

$$O(i, t, p) \rightarrow V_3(t, p) \quad (3)$$

$$S(i, n, e, c) \rightarrow V_4(e) \quad (4)$$

(b) Policy views $\mathcal{V}$

$$P(i, n, e, c) \wedge H_N(i, d) \rightarrow EthDis(e, d) \quad (5)$$

$$P(i, n, e, c) \wedge H_N(i, d) \rightarrow CountyDis(c, d) \quad (6)$$

$$S(i, n, e, c) \wedge O(i, t, p) \rightarrow SO(e) \quad (7)$$

(c) Mapping from S to T.

Figure 2: Schema and tgds in our illustrative example

insurance number and the reason for being admitted to the hospital. Relation O stores information related to patients in oncology departments and, in particular, their insurance numbers, their treatment and their progress. Finally, relation S stores for each student in UK, his insurance number, his name, his ethnicity group and his county.

Consider also the set $\mathcal{V}$ comprising the policy views (1)–(4). The policy views define the information that is safe to make available to the public. View (1) projects the ethnicity groups and the hospital admittance reasons for patients in the north of UK; view (2) projects the counties and the hospital admittance reasons for patients in the south of UK; view (3) projects the treatments and the progress of patients of oncology departments; view (4) projects the ethnicity groups of the school students. The policy views are compliant with the NSS privacy preservation protocol that is adopted at the hospital. Precisely, the NSS privacy preservation protocol considers as unsafe any non-evident piece of information that can potentially de-anonymize an individual. For example, views (1) and (2) do not leak any sensitive information concerning the precise address of patients. Indeed, they include patients from a very large geographical area thus implying that the probability of de-anonymizing a patient is significantly small. Similarly, views (3) and (4) are considered to be safe: the probability of de-anonymizing patients of the oncology department from view (3) is zero, since there is no way to link a patient to his treatment or his progress, while view (4) projects information which is already evident to public.

Finally, consider the following set of source-to-target dependencies Σ_{st}. Dependencies (5) and (6) project similar information with the views (1) and (2), respectively. However, contrarily to the views, they solely focus on patients in the north of UK. Finally, dependency (7) projects the ethnicity groups of students who have been in some oncology department, whereas view V_4 aims at concealing the information about the department in which a student has been admitted.

The above example shows that the policy views defined within one hospital might be in stark contrast with the mappings used to export patient's information to another hospital. This motivates the aforementioned questions (1), (2) and (3) about establishing formal guarantees for privacy preservation as well as enabling repairing of the mappings in order to make them safe. To the best of our knowledge, our work is the first to provide practical algorithms for a logical privacy-preservation paradigm effective in a real system [10], described as an open research challenge in [5, 21]. Our technique is inherently data-independent thus bringing the advantage that both the safety test and the repairing operations are executed on the metadata provided through the mappings and not on the underlying data instances.

The paper is organized as follows. Section 2 discusses the related work. Section 3 presents the basic concepts and notions. Section 4 lays our privacy preservation protocol. Section 5 presents our repairing algorithms and their properties. mechanism. Section 6 outlines the experimental results, while Section 7 concludes our work. The code base along with the experimental data are publicly available at [12].

2 RELATED WORK

Privacy in data integration. Safety of secret queries formulated against a global schema and adhering to the certain answers semantics has been tackled in previous theoretical work [21]. They define the optimal attack that characterizes a set of queries that an attacker can issue to which no further queries can be added to infer more information. They then define the privacy guarantees against the optimal attack by considering the static and the dynamic case, the latter corresponding to modifications of the schemas or the GLAV mappings. The same definition of secret queries and privacy setting is adopted in [5], which instead focuses on boolean conjunctive queries as policy views and on the notion of safety with respect to a given mapping. An ontology-based integration scenario is assumed in which the target instance is produced via a set of mappings starting from an underlying data source. Whereas they study the complexity of the view compliance problem in both data-dependent and data-independent setting, we focus on the latter and extend it to non-boolean conjunctive queries as policy views. We further consider multiple policy views altogether in the design of a practical algorithm for checking the safety of schema mappings and for repairing the mappings in case of violations.

Privacy in data publishing. Data publishing accounts for the settings in which a view exports or publishes the information of an underlying data source. Privacy and information disclosure in data publishing linger over the problem of avoiding the disclosure of the content of the view under a confidential query. A probabilistic formal analysis of the query-view security model has been presented in [20], where they offer a complete treatment of the multi-party collusion and the use of external adversarial knowledge. Access control policies using cryptography are used in [20] to enforce the authorization to an XML document. Our work differs from theirs on both the considered setting, as well as the adopted techniques and the adopted privacy protocol. Striking the balance between utility and privacy in a logic-based framework has been the object of investigation in recent studies focusing on data publishing for Linked Data [13, 14, 19]. The problem there is remarkably different from ours since they focus on publishing a single RDF dataset by applying privacy and utility queries in SPARQL, checking for their compatibility, and for update operations realizing the privacy and utility constraints.

Controlled Query Evaluation. Controlled Query Evaluation is a confidentiality enforcement framework introduced in [23] and refined in [9],[7] and [8], in which a policy declaratively specifies sensitive information and confidentiality is enforced by a censor. Provided a query as input, a censor verifies whether the

query leads to a violation of the policy and in case of a violation it returns a distorted answer. It has been recently adopted in ontologies expressed with Datalog-like rules and in lightweight Description Logics [18]. They assume that the policies are only known to database administrators and not to ordinary users and that the data has protected access through a query interface. Our assumptions and setting are quite different, since our multiple policy views are accessible to every user and our goal is to render the s-t mappings safe with respect to a set of policies via repairing and rewriting.

Data privacy. Previous work has addressed access control to protect database instances at different levels of granularity [22], in order to combine encrypted query processing and authorization rules. Our work being logic-based and declarative does not deal with these authorization methods, as well as does not consider any concrete privacy or anonymization algorithms operating on data instances, such as differential privacy [15] and k-anonymity [24]. Further exploring the connection between concrete privacy enforcement and logic-based privacy formalisms is the subject of future investigation.

Data exchange. The vast literature on data exchange [17] has inspired our work. In the considered scenarios, the source and target schema are considered along with s-t mappings and target dependencies, the latter being both egds and tgds. Similarly, past work on degugging schema mappings [11] has focused on all possible routes generated by the exchange process when incomplete or undefined values in one or more variables are exported from the source instance. By opposite, we focus in this paper on the case in which s-t mappings are coupled with source dependencies under the form of policy views, the latter being typical in privacy scenarios and unexplored in the classical data exchange setting.

3 PRELIMINARIES

Relational symbols and critical instances. Let Const, Nulls, and Vars be mutually disjoint, infinite sets of *constant values*, *labeled nulls*, and *variables*, respectively. A *schema* is a set of *relation names* (or just *relations*), each associated with a nonnegative integer called *arity*. A *relational atom* has the form $R(\vec{t})$ where R is an n-ary relation and $\vec{t}$ is an n-tuple of *terms*, where a term is either a constant, a labelled null, or a variable. An *equality atom* has the form $t_1 = t_2$ where t_1 and t_2 are terms. An atom is called ground or fact, when it does not contain any variables. A position in an n-ary atom A is an integer $1 \leq i \leq n$. We denote by $A|_i$, the i-th term of A. An instance I is a set of relational facts. An atom (resp. an instance) is *null-free* if it does not contain labelled nulls. The *critical instance* of a schema S, denoted as Crt_S, is the instance containing a fact of the form $R(\vec{*})$, for each n-ary relation $R \in \mathsf{S}$, where $*$ is called the *critical constant* and $\vec{*}$ is an n-ary vector. A *substitution* σ is a mapping from variables into constants or labelled nulls.

Dependencies and queries A *dependency* describes the semantic relationship between relations. A *Tuple Generating Dependency* (tgd) is a formula of the form $\forall \vec{x}\ \lambda(\vec{x}) \rightarrow \exists \vec{y}\ \rho(\vec{x}, \vec{y})$, where $\lambda(\vec{x})$ and $\rho(\vec{x}, \vec{y})$ are conjunctions of relational, null-free atoms. An *Equality Generating Dependency* (egd) is a formula of the form $\forall \vec{x}\ \lambda(\vec{x}) \rightarrow x_i = x_j$, where $\lambda(\vec{x})$ is a conjunction of relational, null-free atoms. We usually omit the quantification for brevity. We refer to the left-hand side of a tgd or an egd δ as the *body*, denoted as $\mathsf{body}(\delta)$, and to the right-hand side as the *head*, denoted as $\mathsf{head}(\delta)$. An instance I satisfies a dependency δ, written $I \models \delta$ if each homomorphism from $\mathsf{body}(\delta)$ into I can be extended to a homomorphism h' from $\mathsf{head}(\delta)$ into I. An instance I satisfies a set of dependencies Σ, written as $I \models \Sigma$, if $I \models \delta$ holds, for each $\delta \in \Sigma$. The *solutions* of an instance I w.r.t. Σ is the set of all instances J such that $J \supseteq I$ and $J \models \Sigma$. A solution is called *universal* if it can be homomorphically embedded to each solution of I w.r.t. Σ.

A *conjunctive query* (CQ) is a formula of the form $\exists \vec{y}\ \bigwedge_i A_i$, where A_i are relational, null-free atoms. A CQ is boolean if it does not contain any free variables. A substitution σ is an *answer* to a CQ Q on an instance I if the domain of σ is the free variables of Q, and if σ can be extended to a homomorphism from $\bigwedge_i A_i$ into I. We denote by $Q(I)$, the answers to Q on I.

Let S be a source schema and let T be a target schema. A *mapping* $\mathcal{M}$ from S to T is defined as a triple $(\mathsf{S}, \mathsf{T}, \Sigma)$, where $\Sigma = \Sigma_{st}$;, i.e. the set of the s-t dependencies over S and T. We usually refer to the dependencies in Σ_{st} as *mappings*. A variable x of a mapping $\mu \in \Sigma_{st}$ is called exported if it occurs both in the body and the head of μ. We denote by $\mathsf{exported}(\mu)$, the set of exported variables of μ. The inverse of set of s-t dependencies Σ_{st}, denoted as Σ_{st}^{-1} is the set consisting, for each mapping μ in Σ_{st} of the form $\lambda(\vec{x}) \rightarrow \rho(\vec{x}, \vec{y})$, a mapping μ^{-1} of the form $\rho(\vec{x}, \vec{y}) \rightarrow \lambda(\vec{x})$. We focus on *GAV* mappings in this paper, thus assuming that $\vec{y}$ is empty. Moreover, we consider the setting in which Σ only consists of Σ_{st} thus not including Σ_t. This implies that target egds and target tgds are excluded, since, despite their usage in data exchange, their role is less understood in the privacy-preserving variant considered in this paper.

4 PRIVACY PRESERVATION

In this section, we introduce our notion of privacy preservation. Let $\mathcal{V}$ be a set of *policy views* over S, representing the information that is safe to expose for instances I of S. Our goal is to verify whether a user-defined mapping $\mathcal{M} = (\mathsf{S}, \mathsf{T}, \Sigma)$ is safe w.r.t. the views in a set $\mathcal{V}$. Below, we will introduce a notion for assessing the safety of a GAV mapping $\mathcal{M}_2$ with respect to a GAV mapping $\mathcal{M}_1$, when both make use of the same source schema S. Moreover, let $\Sigma_i = \Sigma_{st_i}$ be the dependencies associated with $\mathcal{M}_i$.

4.1 A formal privacy-preservation protocol

Our notion of privacy preservation builds on the logical foundations introduced in [5] for ontology-based data integration for boolean queries. However, we extend the notion of privacy preservation from [5] to a relational data exchange setting in the presence of non-boolean conjunctive queries. First, we define the notion of disclosure of a CQ by a mapping as follows:

Definition 4.1. A mapping $\mathcal{M} = (\mathsf{S}, \mathsf{T}, \Sigma)$ *does not disclose a CQ* p *over* S *on any instance of* S, if for each instance I of S there exists an instance I' such that $I \equiv_{\mathcal{M}} I'$ and $p(I') = \emptyset$.

The problem of checking whether a mapping $\mathcal{M}$ over S does not disclose a boolean and constants-free CQ p on any instance of S is decidable for GAV mappings consisting of CQ views [5]. In particular, $\mathcal{M}$ does not disclose p on any instance of S if and only if there does not exist a homomorphism from p into the *unique* instance computed by the *visible chase* $\mathsf{visChase}_\mathsf{S}(\Sigma)$ of Σ under the critical instance Crt_S of S. The visible chase computes a *universal source instance* defined as follows:

Definition 4.2. 7 Given a mapping $\mathcal{M} = (\mathsf{S}, \mathsf{T}, \Sigma)$, an instance I is a *universal source instance* over S if for any instance J over the source schema S, there exists an homomorphism h from J into I

such that for any constant c from J that is made visible through $\mathcal{M}$, $h(c) = *$.

The only constant occurring in the instance computed by $\text{visChase}_S(\Sigma)$ is the critical constant $*$ and it represents any other constant that can occur in the source instance.

We introduce our own variant of the visible chase, which organizes the facts derived during chasing into subinstances called *bags*. Algorithm 1 describes the steps of the proposed variant. Please note that Algorithm 1 derives the same set of facts with the algorithm from [5]. However, instead of keeping these facts in a single set, we keep them in separate bags. Before presenting Algorithm 1, we will introduce a couple of useful new notions. The first notion serves the need of defining derived egds that allow to unify as many labeled nulls as possible with the critical constant in the target instance. The second notion allows to define relevant bags for which this unification must hold. Both notions are exploited by the visible chase (Algorithm 1) whose last step triggers the obtained egds.

Definition 4.3. Consider an instance I. Consider also a s-t tgd δ and a homomorphism h from $\text{body}(\delta)$ into I, such that $h(x) \in \text{Nulls}$, for some $x \in \text{exported}(\delta)$. Then, we say that the egd

$$\text{body}(\delta) \rightarrow \bigwedge_{\forall x \in \text{exported}(\delta): h(x) \in \text{Nulls}} x \approx * \qquad (8)$$

is *derived* from δ in I. For an egd ϵ that is derived from a s-t tgd δ in I, $\text{tgd}(\epsilon)$ denotes δ. For a set of s-t tgds Σ and an instance I, $\Sigma_\approx$ is the set comprising for each $\delta \in \Sigma$, the egd that is derived from δ in I.

Definition 4.4. Consider an instance I, whose facts are organized into the bags $\beta_1, \ldots, \beta_m$. Consider also a derived egd δ of the form (8) and an active trigger h for δ in I. A bag β_i is *relevant* for δ and h in I, where $1 \leq i \leq m$, if some fact $F \in h(\text{body}(\delta))$ occurs in β_i and if some $h(x)$ is a labeled null occurring in β_i, where $x \in \text{exported}(\delta)$.

Let $\beta_{j_1}, \ldots, \beta_{j_k} \subseteq \beta_1, \ldots, \beta_m$ be the set of bags that are relevant for δ and h in I. Let $\nu = \{h(x_j) \mapsto h(x_i)\}$ if $h(x_i) = *$, and $\nu = \{h(x_i) \mapsto h(x_j)\}$ if $h(x_i) \notin \text{Const}$, where x_i, x_j are variables from $\text{exported}(\delta)$. Then, the *derived* bag β for δ and h in I consists of the facts in $\bigcup_{l=1}^{k} \nu(\beta_{j_l})$. The bags $\beta_{j_1}, \ldots, \beta_{j_k}$ are called the *predecessors* of β. We use $\beta_{j_l} \prec \beta$ to denote that β_{j_l} is a predecessor of β, for $1 \leq l \leq k$.

We are now ready to proceed with the description of Algorithm 1. Given a s-t mapping, Algorithm 1 computes a universal source instance whose facts are organized into bags. Algorithm 1 first computes the instance I_0 by chasing Crt_S using the s-t tgds, line 1. It then chases I_0 with the inverse s-t tgds Σ^{-1}, line 2. and proceeds by chasing I_1 with the set of all derived egds $\Sigma_\approx$, for each $\delta \in \Sigma$ in I_1, line 4. Algorithm 1 computes a fresh bag at each chase step. In particular, for each active trigger h for δ in I, Algorithm 1 adds a fresh bag with facts $h'(\text{head}(\delta))$, if $\delta \in \Sigma \cup \Sigma^{-1}$, line 9; otherwise, if $\delta \in \Sigma_\approx$, then it adds the derived bag for δ and h in I, see Definition 4.4, line 20.

Note that, $\Sigma_\approx$ aims at "disambiguating" as many labeled nulls occurring in I_1 as possible, by unifying them with the critical constant $*$. Since $*$ represents the information that is "visible" to a third-party, chasing with $\Sigma_\approx$ computes the *maximal information* from the source instance a third-party has access to. Note that Algorithm 1 always terminates [5]. Let $B = \text{visChase}_S(\Sigma)$. We will denote by $I_S(\Sigma)$, the instance $\bigcup_{\beta \in B} \beta$.

Algorithm 1 $\text{visChase}_S(\Sigma)$

1: $B_0 := \text{bagChaseTGDs}(\Sigma, \text{Crt}_S)$
2: $B_1 := \text{bagChaseTGDs}(\Sigma^{-1}, \bigcup_{\beta \in B_0} \beta \setminus \text{Crt}_S)$
3: Let $\Sigma_\approx$ be the set of all derived egds $\Sigma_\approx$, for each $\delta \in \Sigma$ in I_1
4: **return** $\text{bagChaseEGDs}(\Sigma_\approx, B_0 \cup B_1)$

5: **procedure** $\text{bagChaseTGDs}(\Sigma, I)$
6: $B := \emptyset$
7: **for each** $\delta \in \Sigma$ **do**
8: **for each** active trigger $h : \text{body}(\delta) \rightarrow I$ **do**
9: **create** a fresh bag β with facts $h'(\text{head}(\delta))$
10: **add** β to B
11: **return** B

12: **procedure** $\text{bagChaseEGDs}(\Sigma_\approx, B)$
13: $i := 0$; $I_i := \bigcup_{\beta \in B} \beta$
14: **do**
15: $i := i + 1$
16: **for each** ($\delta \in \Sigma_\approx$ of the form (8) **do**
17: **for each** active trigger $h : \text{body}(\delta) \rightarrow I_{i-1}$ **do**
18: **if** $h(x) \neq *$, for some $x \in \text{exported}(\delta)$ **then**
19: Let β be the derived bag for δ and h in I_{i-1}
20: **add** β to B
21: $I_i := I_i \cup \beta$
22: **while** $I_{i-1} \neq I_i$
23: **return** B

Example 4.5. We demonstrate the visible chase algorithm over the policy views and the s-t dependencies from Example 1.1. We show how the algorithm runs first on the policy views $\mathcal{V}$ and then show the computation on Σ_{st}.

We first present the computation of $I_S(\mathcal{V}) = \bigcup_{\beta \in \text{visChase}_S(\mathcal{V})} \beta$.

The critical instance Crt_S of S consists of the facts shown in the following Eq. (9)

$$\text{P}(*,*,*,*) \quad \text{H}_\text{N}(*,*) \quad \text{H}_\text{S}(*,*) \quad \text{O}(*,*,*) \quad \text{S}(*,*,*,*) \qquad (9)$$

where $*$ is the critical constant.

The instance I_1 computed by chasing the output of line 1 using $\mathcal{V}^{-1}$ will consist of the facts

$$\begin{array}{lll} \text{P}(\text{n}_\text{i}, \text{n}_\text{n}, *, \text{n}_\text{c}) & \text{H}_\text{N}(\text{n}_\text{i}, *) & \text{O}(\text{n}''_\text{i}, *, *) \\ \text{P}(\text{n}'_\text{i}, \text{n}'_\text{n}, \text{n}_\text{e}, *) & \text{H}_\text{S}(\text{n}'_\text{i}, *) & \text{S}(\text{n}'''_\text{i}, \text{n}'''_\text{n}, *, \text{n}'''_\text{c}) \end{array} \qquad (I_1)$$

where the constants prefixed by n are labeled nulls created while chasing Crt_S with the inverse mappings. Since there exists no homomorphism from the body of any s-t tgd into I_1 mapping an exported variable into a labeled null, $\Sigma_\approx$ will be empty, see Definition 4.3. Thus, $I_S(\mathcal{V}) = I_1$.

We next present the computation of $I_S(\Sigma_{st}) = \bigcup_{\beta \in \text{visChase}_S(\Sigma_{st})} \beta$. The instance I'_1 computed by chasing the output of line 1 by Σ_{st}^{-1} will consist of the facts

$$\begin{array}{lll} \text{P}(\text{n}_\text{i}, \text{n}_\text{n}, *, \text{n}_\text{c}) & \text{H}_\text{N}(\text{n}_\text{i}, *) & \text{S}(\text{n}''_\text{i}, \text{n}''_\text{n}, *, \text{n}'_\text{c}) \\ \text{P}(\text{n}'_\text{i}, \text{n}'_\text{n}, \text{n}_\text{e}, *) & \text{H}_\text{N}(\text{n}'_\text{i}, *) & \text{O}(\text{n}''_\text{i}, \text{n}''_\text{t}, \text{n}''_\text{p}) \end{array} \qquad (I'_1)$$

Since there exists a homomorphism from the body of μ_e into I'_1 mapping the exported variable e into the labeled null n_e, and since there exists another homomorphism from the body of μ_c into I'_1 mapping the exported variable c into the labeled null n_c, $\Sigma_\approx$ will comprise the egds ϵ_1 and ϵ_2 shown below

$$\text{P}(\text{i}, \text{n}, \text{e}, \text{c}) \wedge \text{H}_\text{N}(\text{i}, \text{d}) \rightarrow \text{e} \approx * \qquad (\epsilon_1)$$

$$\text{P}(\text{i}, \text{n}, \text{e}, \text{c}) \wedge \text{H}_\text{N}(\text{i}, \text{d}) \rightarrow \text{c} \approx * \qquad (\epsilon_2)$$

The last step of the visible chase involves chasing I_1' using $\Sigma_{\approx}$. W.l.o.g, assume that the chase considers first ϵ_1 and then ϵ_2. During the first step of the chase, there exists a homomorphism from body(ϵ_1) into I_1'. Hence, $n_e = *$. During the second step of the chase, there exists a homomorphism from body(ϵ_2) into I_1' and, hence, $n_c = *$. The instance computed at the end of the second round of the chase will consist of the facts

$$\begin{array}{llll} P(n_i, n_n, *, *) & H_N(n_i, *) & H_N(n_i', *) & (10) \\ S(n_i'', n_n'', *, n_c') & O(n_i'', n_t'', n_p'') & & \end{array}$$

Since there exists no active trigger for ϵ_1 or ϵ_2 in the above instance (Eq (10)), the chase will terminate.

The facts in $I_S(\Sigma_{st})$ will be organized into the following bags β_1–β_5 (one bag per line)

$$\begin{aligned} SO(e) &\xrightarrow{\langle \mu_s^{-1}, h_1 \rangle} S(n_i'', n_n'', *, n_c'), O(n_i'', n_t'', n_p'') \\ CountyDis(c, d) &\xrightarrow{\langle \mu_c^{-1}, h_2 \rangle} P(n_i', n_n', n_e, *), H_N(n_i', *) \\ EthDis(e, d) &\xrightarrow{\langle \mu_e^{-1}, h_3 \rangle} P(n_i, n_n, *, n_c), H_N(n_i, *) \\ P(n_i', n_n', n_e, *), H_N(n_i', *) &\xrightarrow{\langle \epsilon_1, h_4 \rangle} P(n_i', n_n', *, *), H_N(n_i', *) \\ P(n_i, n_n, *, n_c), H_N(n_i, *) &\xrightarrow{\langle \epsilon_2, h_5 \rangle} P(n_i, n_n, *, *), H_N(n_i, *) \end{aligned}$$

$$\begin{aligned} h_1 &= \{i \mapsto n_i', n \mapsto n_n', e \mapsto n_e, c \mapsto *, d \mapsto *\} \\ h_2 &= \{c \mapsto *, d \mapsto *\} \\ h_3 &= \{e \mapsto *, d \mapsto *\} \\ h_4 &= \{i \mapsto n_i', n \mapsto n_n', e \mapsto n_e, c \mapsto *, d \mapsto *\} \\ h_5 &= \{i \mapsto n_i, n \mapsto n_n, e \mapsto *, c \mapsto n_c, d \mapsto *\} \end{aligned}$$

The contents of the bags correspond to the right-hand side of the arrows. However, for presentation purposes, we also show the related dependency δ and the homomorphism h that lead to the derivation of each bag (shown at the top of each arrow), as well as, the facts in h(body(δ)) (left-hand side of each arrow). The obtained bags will be part of the universal source instance $I_S(\Sigma_{st})$. Such an instance will be used in Section 5 in order to apply the notion of safety in the repairing of the underlying mappings Σ_{st}.

4.2 Preserving the privacy of policy views

We consider a mapping $\mathcal{M} = (S, T, \Sigma)$ to be safe w.r.t. a view mapping $\mathcal{M}_V = (S, V, \mathcal{V})$ (with $\mathcal{V}$ being the set of policy views and V being the schema of the views as shown in Figure 1), if $\mathcal{M}$ does not disclose the information that is also not disclosed by $\mathcal{M}_V$. Definition 4.6 and Theorem 4.7 presented below formalize our notion of privacy preservation and show that there exists a simple process for verifying whether $\mathcal{M}$ is safe w.r.t. $\mathcal{M}_V$.

Definition 4.6. A mapping $\mathcal{M}_2 = (S, T_2, \Sigma_2)$ *preserves the privacy of a mapping* $\mathcal{M}_1 = (S, T_1, \Sigma_1)$ *on all instances of* S, if for each constants-free CQ p over S, if p is not disclosed by $\mathcal{M}_1$ on any instance of S, then p is not disclosed by $\mathcal{M}_2$ on any instance of S.

Theorem 4.7. *A mapping* $\mathcal{M}_2 = (S, T_2, \Sigma_2)$ preserves the privacy of a mapping $\mathcal{M}_1 = (S, T_1, \Sigma_1)$ on all instances of S, *if and only if there exists a homomorphism* h *from* $I_S(\Sigma_2)$ *into* $I_S(\Sigma_1)$, *such that* $h(*) = *$.

Proof. (Sketch) First we show that the following holds:

Lemma 4.8. *A mapping* $\mathcal{M} = (S, T, \Sigma)$ does not disclose a constants-free CQ p over S on any instance of S, *iff* $\vec{*} \notin p(J)$, *where* $J = I_S(\Sigma_{st})$.

Proof. By adapting the proof technique of Theorem 16 from [5], we can show that $J = I_S(\Sigma_{st})$ is a *universal* source instance $I_S(\Sigma)$ satisfying the following property: for each pair of source instances I and I', such that I' is indistinguishable from I w.r.t. the mapping $\mathcal{M}$, there exists a homomorphism h from I' into $I_S(\Sigma)$ mapping each schema constant into the critical constant $*$. Due to the existence of a homomorphism h from I' into $I_S(\Sigma)$, for each pair of indistinguishable source instances I and I', we can see that if $\vec{*} \notin p(J)$ for a constants-free CQ p, then $p(I') = \emptyset$. Due to the above and due to Definition 4.1, it follows that $\mathcal{M} = (S, T, \Sigma)$ does not disclose a constants-free CQ p over S on any instance of S. □

Lemma 4.8 states that, in order to check if a constants-free CQ is safe according to Definition 4.1, we need to check if the critical tuple is among the answers to p over the instance computed by visChase$_S(\Sigma)$. Next, we show the following lemma.

Lemma 4.9. *Given two instances* I_1 *and* I_2, *the following are equivalent*

(1) for each CQ p, *if* $\vec{u} \in p(I_1)$, *then* $\vec{u} \in p(I_2)$, *where* $\vec{u}$ *is a vector of constants*
(2) there exists a homomorphism from I_1 *to* I_2 *preserving the constants of* I_1

Proof of Lemma 4.9. (2)⇒(1). Suppose that there exists a homomorphism h from I_1 to I_2 preserving the constants of I_1. Suppose also that $\vec{u} \in p(I_1)$, with p being a CQ. This means that there exists a homomorphism h_1 from p into I_1 mapping each free variable x_i of p into u_i, for each $1 \leq i \leq n$, where n is the number of free variables of p. Since the composition of two homomorphisms is a homomorphism and since h preserves the constants of I_1 due to the base assumptions, this means that $h \circ h_1$ is a homomorphism from p into I_2 mapping each free variable x_i of p into t_i, for each $1 \leq i \leq n$. This completes this part of the proof.

(1)⇒(2). Let p_1 be a CQ formed by creating a non-ground atom $R(y_1, \ldots, y_n)$ for each ground atom $R(u_1, \ldots, u_n) \in I_1$, by taking the conjunction of these non-ground atoms and by converting into an existentially quantified variable every variable created out of some labelled null. Let $\vec{x}$ denote the free variables of p_1 and let $n = |\vec{x}|$. From the above, it follows that there exists a homomorphism h_1 from p_1 into I_1 mapping each $x_i \in \vec{x}$ into some constant occurring in I_1. Let $\vec{u} \in p_1(I_1)$. From (1), it follows that $\vec{u} \in p_1(I_2)$ and, hence, there exists a homomorphism h_2 from p_1 into I_2 mapping each $x_i \in \vec{x}$ into u_i, for each $1 \leq i \leq n$. Since h_1 ranges over all constants of I_1 and since $h_1(x_i) = h_2(x_i)$ holds for each $1 \leq i \leq n$, it follows that there exists a homomorphism from I_1 to I_2 preserving the constants of I_1. This completes the second part of the proof. □

Lemma 4.9 can be restated as follows:

Lemma 4.10. *Given two instances* I_1 *and* I_2, *the following are equivalent*

(1) for each CQ p, *if* $\vec{t} \notin p(I_2)$, *then* $\vec{t} \notin p(I_1)$
(2) there exists a homomorphism from I_1 *to* I_2

We are now ready to return to the main part of the proof. Given a CQ p over a source schema S, and a mapping $\mathcal{M}$ defined as the triple (S, T, Σ), where T is a target schema and Σ is a set of s-t dependencies, we know from Proposition 4.8 that if $\mathcal{M}$ discloses p on some instance of S, then there exists a homomorphism of p into visChase$_S(\Sigma)$ mapping the free variables of p into the critical constant $*$.

From the above, we know that $\mathcal{M}_2$ does not preserve the privacy of $\mathcal{M}_1$ if there exists a CQ p over S, such that $\vec{*} \notin J_1$ and $\vec{*} \in J_2$, where $J_1 = I_S(\Sigma_1)$ and $J_2 = I_S(\Sigma_2)$. We will now prove that $\mathcal{M}_2$ preserves the privacy of $\mathcal{M}_1$ iff there exists a homomorphism from J_2 into J_1 that preserves the critical constant $*$. This will be referred to as conjecture (C).

($\Rightarrow$) If $\mathcal{M}_2$ preserves the privacy of $\mathcal{M}_1$, then for each CQ p, if $\vec{*} \notin p(J_1)$, then $\vec{*} \notin p(J_2)$. From the above and from Lemma 4.10, it follows that there exists a homomorphism $\phi : J_2 \rightarrow J_1$, such that $\phi(*) = *$.

($\Leftarrow$) The proof proceeds by contradiction. Assume that there exists a homomorphism h from J_2 into J_1 preserving $*$, but $\mathcal{M}_2$ does not preserve the privacy of $\mathcal{M}_1$. We will refer to this assumption as assumption (A_1). From assumption (A_1) and the discussion above it follows that there exists a CQ p over S such that $\vec{*} \notin p(J_1)$ and $\vec{*} \in p(J_2)$. Let h_2 be the homomorphism from p into J_2 mapping its free variables into $*$. Since the composition of two homomorphisms is a homomorphism, this means that $h \circ h_2$ is a homomorphism from p into J_1 mapping its free variables into $*$, i.e., $\vec{*} \in p(J_1)$. This contradicts our original assumption and hence concludes the proof of conjecture (C). Conjecture (C) witnesses the decidability of the instance-independent privacy preservation problem: in order to verify whether $\mathcal{M}_2$ preserves the privacy of $\mathcal{M}_1$ we only need to check if there exists a homomorphism $\phi : I_S(\Sigma_2) \rightarrow I_S(\Sigma_1)$, such that $\phi(*) = *$. □

Theorem 4.7 states that in order to verify whether $\mathcal{M}_2$ is safe w.r.t. $\mathcal{M}_1$, we need to compute $I_S(\Sigma_1)$ and $I_S(\Sigma_2)$ and check if there exists a homomorphism from the second instance into the first one that maps $*$ into itself. If there exists such a homomorphism, we say that $I_S(\Sigma_1)$ is *safe* w.r.t. $I_S(\Sigma_2)$, or simply safe, and we say that it is unsafe otherwise.

Example 4.11. Continuing with Example 1.1, we can see that the s-t tgds are not safe w.r.t. the policy views according to Theorem 4.7, since there does not exist a homomorphism from the instance $I_S(\Sigma_{st})$ into the instance $I_S(\mathcal{V})$. This means that there exists information which is disclosed by Σ_{st} in some instance that satisfies Σ_{st}, but it is not disclosed by $\mathcal{V}$. Indeed, from $S(n_i'', n_n'', *, n_c')$ and $O(n_i'', n_t'', n_p'')$, we can see that we can potentially leak the identity of a student who has been to an oncology department. This can happen if there exists only one student in the school coming from a specific ethnicity group and this ethnicity group is returned by μ_s. Please note that the policy views are safe w.r.t. this leak. Indeed, it is impossible to derive this information through reasoning over the returned tuples under the input instance and the views V_3 and V_4.

Furthermore, by looking at the facts $P(n_i, n_n, *, *)$ and $H_N(n_i, *)$, we can see that we can potentially leak the identity and the disease of a patient who has been admitted to some hospital in the north of UK. This can happen if there exists only one patient who relates to the county and the ethnicity group returned by μ_e and μ_c. Note that the policy views V_1 and V_2 do not leak this information, since it is impossible to obtain the county and the ethnicity group of an NHS patient at the same time.

5 REPAIRING UNSAFE MAPPINGS

In Section 4, we presented our privacy preservation protocol and a technique for verifying whether a mapping is safe w.r.t. another one, over all source instances. This section presents an algorithm for repairing an unsafe mapping $\mathcal{M}$ w.r.t. a set of policy views $\mathcal{V}$. This is a fundamental operation needed to amend mappings whenever the policy views are modified and become unsafe (e.g. in the presence of data protection regulations).

Algorithm 2 repair$(\Sigma, \mathcal{V}, \text{prf}, n)$

1: $\Sigma_1 :=$ frepair$(\Sigma, \mathcal{V}, \text{prf})$
2: $\Sigma_2 :=$ srepair$(\Sigma_1, \mathcal{V}, \text{prf}, n)$
3: **return** Σ_2

Algorithm 2 summarizes the steps of the proposed algorithm. The inputs to it are, apart from Σ and $\mathcal{V}$, a positive integer n which will be used during the second step of the repairing process and a preference mechanism prf for ranking the possible repairs. In the simplest scenario, the preference mechanism implements a fixed function for ranking the different repairs. However, it can also employ supervised learning techniques in order to progressively learn the user preferences by looking at his prior decisions.

Since a mapping $\mathcal{M}$ is safe w.r.t. $\mathcal{V}$ if the instance $I_S(\Sigma)$ is safe according to Theorem 4.7, Algorithm 2 rewrites the tgds in $\mathcal{M}$, such that the derived visible chase instances are safe. The rewriting takes place in two steps. The first step rewrites Σ into a *partially-safe* set of s-t dependencies Σ_1, while the second step rewrites the output of the first one into a new set of s-t dependencies Σ_2, such that $I_S(\Sigma_2)$ is safe. As we will explain later on, partial-safety ensures that the intermediate instance I_1 produced by visChase$_S(\Sigma_1)$ at line 2 of Algorithm 1 is safe, but it does not provide strong privacy guarantees. The benefit of this two-step approach is that it allows repairing one or a small set of dependencies at a time.

5.1 Computing partially-safe mappings

Since the problem of safety is reduced to the problem of checking for a homomorphism from $I_S(\Sigma)$ into $I_S(\mathcal{V})$, a first test towards checking for such a homomorphism is to look if the mappings in Σ would lead to such a homomorphism or not. For instance, by looking at μ_s in Example 1.1 it is easy to see that it leaks sensitive information, since it involves a join between students and oncology departments, which does not occur in $I_S(\mathcal{V})$.

Definition 5.1. A mapping $\mathcal{M} = (S, T, \Sigma)$ is *partially-safe w.r.t.* $\mathcal{M}_V = (S, V, \mathcal{V})$ *on all instances of* S, if there exists a homomorphism from chase$(\Sigma^{-1}, \text{Crt}_T) \setminus \text{Crt}_T$ into $I_S(\mathcal{V})$.

From Algorithm 1, it follows that Σ is partially-safe iff the intermediate instance I_1 computed by visChase$_S(\Sigma)$ is safe.

Proposition 5.2. *A mapping* $\mathcal{M} = (S, T, \Sigma)$ *is* partially-safe w.r.t. $\mathcal{M}_V = (S, V, \mathcal{V})$ on all instances of S, *if for each* $\mu \in \Sigma$, *there exists a homomorphism from* body(μ) *into* $I_S(\mathcal{V})$ *mapping each* $x \in$ exported(μ) *into the critical constant* $*$.

Note that according to Proposition 5.2, in our running example Σ_{st} would be partially-safe, if $\mu_s \notin \Sigma_{st}$, then since there exist homomorphisms from the bodies of μ_s and μ_c into $I_S(\mathcal{V})$, mapping their exported variables into $*$. It is also easy to show the following

Remark 1. *A mapping* $\mathcal{M} = (S, T, \Sigma)$ *is* safe w.r.t. $\mathcal{M}_V = (S, V, \mathcal{V})$ on all instances of S, *only if it is partially-safe w.r.t.* $\mathcal{M}_V$ *on all instances of* S. □

Proposition 5.2 presents a quite convenient, yet somewhat expected, finding: in order to obtain a partially-safe mapping, it suffices to repair each s-t dependency *independently of the others*. Furthermore, the repair of each $\mu \in \Sigma$ involves breaking joins

and hiding exported variables, such that the repaired dependency μ_r satisfies the criterion in Proposition 5.2.

We make use of the result of Proposition 5.2 in Algorithm 3. Algorithm 3 obtains, for each $\mu \in \Sigma$, a set of rewritings $\mathcal{R}_\mu$, out of which we will choose the best rewriting according to prf. The set $\mathcal{R}_\mu$ consists of *all* rewritings that differ from μ w.r.t. the variable repetitions in the bodies of the rules and the exported variables. Below, we present the steps of Algorithm 3.

For each s-t tgd μ and for each atom $B \in \text{body}(\mu)$, Algorithm 3 constructs a fresh atom C and adds C to a set $\mathcal{C}$. The set of atoms $\mathcal{C}$ provides us with the means to identify all repairs of μ that involve breaking joins and hiding exported variables. In particular, each homomorphism ξ from $\mathcal{C}$ into $I_S(\mathcal{V})$ corresponds to one repair of μ. In lines 12–25, Algorithm 3 modifies each atom $B \in \text{body}(\mu)$ by taking into account prior body atom modifications. The prior modifications are accumulated in the relation ρ and the mapping ψ. The relation ρ keeps for each variable x from $\text{body}(\mu)$, the fresh variables that were used to replace x during prior steps of the repairing process, while ψ is a substitution from the partially repaired body into $I_S(\mathcal{V})$. In particular, at the end of the i-th iteration of the loop in line 12, ψ holds the substitution from the first repaired i atoms from $\text{body}(\mu)$ into $I_S(\mathcal{V})$. We adopt this approach instead of replacing variable x in position p always by a fresh variable, in order to minimize the number of the joins we break.

Below, we describe how Algorithm 3 modifies each body atom of μ, w.r.t. a homomorphism ξ, lines 9–27. Let $C = \nu(B)$ be the fresh body atom that was constructed out of B in line 5. For each atom $B \in \text{body}(\mu)$ and for each $p \in \text{pos}(B)$, if the variable y in position p of C is not mapped to the critical constant $*$ via ξ and $B|_p$ is an exported variable, this means that *the variable sitting in position p of B should not be exported* (see first condition in line 16). Similarly, if the variable sitting in position p of B is mapped to a different constant than the one that y maps via ξ, then this means that *the variable sitting in position p of B introduces an unsafe join* (see second condition in line 16). In the presence of these violations, we must replace variable x in position p of B, either by a variable that was used in a prior step of the repairing process, line 17–18), or by a fresh variable, lines 19–23. Otherwise, if there is no violation so far, then we add the mapping $\{x \mapsto \xi(y)\}$ to ψ, if it is not already there, lines 24–25. Finally, the algorithm chooses the best repair according to the preference function, lines 28–31.

Algorithm 3 frepair$(\Sigma, \mathcal{V}, \text{prf})$

```
1:  for each μ ∈ Σ do
2:      ν := ∅, C := ∅
3:      for each B ∈ body(μ), where B = R(x⃗) do
4:          create a vector of fresh variables y⃗
5:          create the atom C = R(y⃗)
6:          add (B, C) to ν
7:          add C to C
8:      R_μ := ∅
9:      for each homomorphism ξ : C → I_S(V) do
10:         ρ := ∅, ψ := ∅
11:         μ_r := μ
12:         for each B ∈ body(μ_r) do
13:             C = ν(B)
14:             for each p ∈ pos(B) do
15:                 x = B|_p, y = C|_p
16:                 if x ∈ exported(μ) and * ≠ ξ(y) or x ∈
    dom(ψ) and ψ(x) ≠ ξ(y) then
17:                     if ∃x' s.t. (x, x') ∈ ρ and ψ(x') = ξ(y)
    then
18:                         B|_p = x'
19:                     else
20:                         create a fresh variable x'
21:                         add (x, x') to ρ
22:                         add {x' ↦ ξ(y)} to ψ
23:                         B|_p = x'
24:                 else if x ∉ dom(ψ) then
25:                     add {x ↦ ξ(y)} to ψ
26:         if μ_r ≠ μ then
27:             add μ_r to R_μ
28:     if R_μ ≠ ∅ then f
29:         choose the best repair μ_r of μ from R_μ based on prf
30:         remove μ from Σ
31:         add μ_r to Σ
32: return Σ
```

Proposition 5.3. *For any $\mathcal{M} = (S, T, \Sigma)$, any $\mathcal{M}_V = (S, V, \mathcal{V})$ and any preference function* prf*, Algorithm* frepair *returns a mapping $\mathcal{M}' = (S, T, \Sigma')$ that is partially-safe w.r.t. $\mathcal{M}_V$ on all instances of S.*

Proof. (Sketch) From Proposition 5.2, a mapping $\mathcal{M} = (S, T, \Sigma)$ is partially-safe w.r.t. $\mathcal{M}_V = (S, V, \mathcal{V})$ on all instances of S, if for each $\mu \in \Sigma$, there exists a homomorphism from $\text{body}(\mu)$ into $I_S(\mathcal{V})$ mapping each $x \in \text{exported}(\mu)$ into the critical constant $*$. Since for each $\mu \in \Sigma$ frepair computes a set of repaired tgds $\mathcal{R}_\mu$, it follows that Proposition 5.3 holds, if such a homomorphism exists, for each repaired tgd in $\mathcal{R}_\mu$. The proof proceeds as follows. Let μ_r^i and ψ^i denote the repaired s-t tgd and the homomorphism ψ computed at the end of each iteration i of the steps in lines 12–25 of Algorithm 3. Let also B^i denote the i-th atom in $\text{body}(\mu_r)$. Since each $C \in \mathcal{C}$ is an atom of distinct fresh variables, since ξ is a homomorphism from $\mathcal{C}$ to $I_S(\mathcal{V})$ and since $\psi(B^i) = \mu_r|_i$, it follows that in order to prove Proposition 5.2, we have to show that the following claim holds, for each $i \geq 0$:

- ϕ. ψ^i is a homomorphism from the first i atoms in the body of μ_r into $I_S(\mathcal{V})$ mapping each exported variable occurring in $B^0, \ldots, B^i$ into the critical constant $*$.

For $i = 0$, ϕ trivially holds. For $i + 1$ and assuming that ϕ holds for i let $C^{i+1} = \nu(B^{i+1})$, line 13. The proof of claim ϕ depends upon the proof of the following claim, for each iteration $p \geq 0$ of the steps in lines 14–25:

- θ. $\psi^{i+1}(B^{i+1}|_p) = \xi(y)$, where $y = C^{i+1}|_p$.

The claim θ trivially holds for $p = 0$, while for $p > 0$, it directly follows from the steps in lines 16–25. Since ϕ holds for i, since the steps in lines 16–25 do not modify the variable mappings in ψ^i and due to θ, it follows that ϕ holds for $i + 1$, concluding the proof of Proposition 5.3. □

Example 5.4. We demonstrate an example of Algorithm 3.

Since Algorithm 3 focuses on $I_S(\mathcal{V})$ overlooking the actual views in $\mathcal{V}$, we will not explicitly define $\mathcal{V}$. Instead, we will only assume that the visible chase computes the instance

$$I_S(\mathcal{V}) = \{R_1(*, n_1, n_2), S_1(n_1, n_2, n_2), S_1(n_1, n_3, *), S_1(n_1, *, *)\}$$

where n_1–n_3 are labeled nulls. Consider also the mapping $\mathcal{M}$ consisting of the following s-t dependency

$$R_1(x, y, z) \wedge S_1(y, z, z) \rightarrow T_1(x, z) \qquad (\mu_1)$$

Note that $\mathcal{M}$ is not partially-safe. Algorithm 3 computes two repairs for μ_1 by applying the steps described below. First, it computes the atoms $\mathsf{R}_1(x_1, x_2, x_3)$ $\mathsf{S}_1(x_4, x_5, x_6)$ and adds them to C, lines 3–7. Then, it identifies the following three homomorphisms from C into $I_{\mathsf{S}}(\mathcal{V})$:

$$\xi_1 = \{x_1 \mapsto *, x_2 \mapsto \mathsf{n}_1, x_3 \mapsto \mathsf{n}_2, x_4 \mapsto \mathsf{n}_1, x_5 \mapsto \mathsf{n}_2, x_6 \mapsto \mathsf{n}_2\}$$
$$\xi_2 = \{x_1 \mapsto *, x_2 \mapsto \mathsf{n}_1, x_3 \mapsto \mathsf{n}_2, x_4 \mapsto \mathsf{n}_1, x_5 \mapsto \mathsf{n}_3, x_6 \mapsto *\}$$
$$\xi_3 = \{x_1 \mapsto *, x_2 \mapsto \mathsf{n}_1, x_3 \mapsto \mathsf{n}_2, x_4 \mapsto \mathsf{n}_1, x_5 \mapsto *, x_6 \mapsto *\}$$

From ξ_1, we can see that the joins in the body of μ_1 are safe; however, it is unsafe to export z. From ξ_2, we can see that is safe to reveal the third position of S_1; however, it is unsafe to join the second and the third position of S_1. Algorithm 3 then iterates over ξ_1 and ξ_2, line 9. When $B = \mathsf{R}_1(x, y, z)$ and $p < 3$, Algorithm 3 computes ψ to $\{x \mapsto *, y \mapsto \mathsf{n}_1\}$, since there is no violation according to line 16. When $B = \mathsf{R}_1(x, y, z)$ and $p = 3$, however, a violation is detected. This is due to the fact that z is an exported variable and $\xi(x_3) = \mathsf{n}_2$. Algorithm 3 tackles this violation by creating a fresh variable z_1. Then, it adds the relation (z, z_1) to ρ, replaces z in $B|_3$ by z_1 and adds the mapping $\{z_1 \mapsto \mathsf{n}_2\}$ to ψ, lines 19–23. Algorithm 3 then considers $\mathsf{S}_1(y, z, z)$. When $p = 1$, no violation is encountered, since $\psi(y) = \xi_1(x_4)$. However, when $p = 2$, a homomorphism violation is encountered, since z is an exported variable and since $\xi(x_3) = \mathsf{n}_2$. Since $(z, z_1) \in \rho$ and $\psi(z_1) = \xi_1(x_5)$, Algorithm 3 replaces z in the second position of $\mathsf{S}_1(y, z, z)$ by z_1, line 19. By applying a similar reasoning, we can see that the variable z siting in $\mathsf{S}_1(y, z, z)|_3$ is also replaced by z_1. Hence, the first repair of μ is

$$\mathsf{R}_1(x, y, z_1) \wedge \mathsf{S}_1(y, z_1, z_1) \rightarrow \mathsf{T}_1(x) \qquad (r_1)$$

Algorithm 3, then proceeds by repairing μ_1 based on ξ_2. When $B = \mathsf{R}_1(x, y, z)$, Algorithm 3 proceeds as described above and computes ψ to $\{x \mapsto *, y \mapsto \mathsf{n}_1, z_1 \mapsto \mathsf{n}_2\}$. When $B = \mathsf{S}_1(y, z, z)$ and $p = 1$, then no violation is encountered since $\psi(y) = \xi_1(x_4)$, while when $B = \mathsf{S}_1(y, z, z)$ and $p = 2$, there is a violation. Since the condition in line 18 is not met, Algorithm 3 creates a fresh variable z_2 and adds the mapping $\{z_2 \mapsto \mathsf{n}_3\}$ to ψ. When $B = \mathsf{S}_1(y, z, z)$ and $p = 3$, then no violation is met, since $z \in \text{exported}(\mu)$ and $\xi_2(x_6) = *$. Hence, the second repair of μ_1 is

$$\mathsf{R}_1(x, y, z_1) \wedge \mathsf{S}_1(y, z_2, z) \rightarrow \mathsf{T}_1(x, z) \qquad (r_2)$$

Finally, we can see that the repair for μ_1 w.r.t. ξ_3 is

$$\mathsf{R}_1(x, y, z_1) \wedge \mathsf{S}_1(y, z, z) \rightarrow \mathsf{T}_1(x, z) \qquad (r_3)$$

5.2 Computing safe mappings

Unifications of one or more labeled nulls occurring in I_1 with the critical constant $*$, might lead to unsafe instances. Consider, for instance, a simplified variant of Example 1.1, where Σ_{st} comprises only μ_e and μ_c. Both μ_e and μ_c are partially-safe, as we have explained above. However, the unification of the labeled nulls n_n and n_c produces an unsafe instance. Algorithm 4 aims at repairing the output of the previous step, such that no unsafe unification of a labeled null with $*$ takes place.

Consider again the simplified variant of Σ_{st} from above. Since Σ_{st} is partially-safe, it suffices to look for homomorphism violations in I_i, for $i \geq 1$. A first observation is that the homomorphism violations are "sitting" within the bags. This is due to the fact that each bag stores *all* the facts associated with the bodies of one or more s-t tgds from Σ_{st}. A second observation is that one way for preventing unsafe unifications is to hide exported variables. For example, let us focus on the unsafe unification of n_e with $*$. This unification takes place due to ϵ_1, which in turn has been created due to the fact that e is an exported variable in μ_e. By hiding the exported variable e from μ_e, we actually prevent the creation of ϵ_1 and hence, we block the unsafe unification of e with $*$. Hiding exported variables is one way for preventing unsafe unifications with the critical constant. Another way for preventing unsafe unifications is to break joins in the bodies of the rules.

Example 5.5. This example demonstrates a second approach for preventing unsafe labeled null unifications.

Consider a set of policy views $\mathcal{V}$ leading to the following instance $I_{\mathsf{S}}(\mathcal{V}) = \{\mathsf{R}_1(\mathsf{n}_1, \mathsf{n}_1, *), \mathsf{R}_1(*, *, \mathsf{n}_2), \mathsf{S}_1(*)\}$, where n_1 and n_2 are labelled nulls. Consider also the mapping $\mathcal{M}$ consisting of the following s-t dependencies:

$$\mathsf{R}_1(x, x, y) \wedge \mathsf{S}_1(y) \rightarrow \mathsf{T}_1(y) \qquad (\mu_2)$$
$$\mathsf{R}_1(x, x, y) \rightarrow \mathsf{T}_2(x) \qquad (\mu_3)$$

It is easy to see that $\mathcal{M}$ is partially-safe, but unsafe in overall. Indeed, $I_{\mathsf{S}}(\Sigma)$ will consist of the following bags (for presentation purposes, we adopt the notation from Example 4.5):

$$\mathsf{T}_1(*) \xrightarrow{\langle \mu_2^{-1}, \theta_1 \rangle} \mathsf{R}_1(\mathsf{n}_3, \mathsf{n}_3, *), \mathsf{S}_1(*)$$
$$\mathsf{T}_2(*) \xrightarrow{\langle \mu_3^{-1}, \theta_2 \rangle} \mathsf{R}_1(*, *, \mathsf{n}_4)$$
$$\mathsf{R}_1(\mathsf{n}_3, \mathsf{n}_3, *), \mathsf{S}_1(*) \xrightarrow{\langle \epsilon_3, \theta_3 \rangle} \mathsf{R}_1(*, *, *), \mathsf{S}_1(*)$$

where $\epsilon_3 := \mathsf{R}_1(x, x, y) \rightarrow x = *, \theta_1 = \{y \mapsto *\}, \theta_2 = \{x \mapsto *\}$ and $\theta_3 = \{x \mapsto \mathsf{n}_3, y \mapsto *\}$. Note that ϵ_3 has been created out of μ_3, since there exists a homomorphism from $\text{body}(\mu_3)$ into $\mathsf{R}_1(\mathsf{n}_3, \mathsf{n}_3, *)$ mapping the exported variable x into n_3.

One approach for preventing the unsafe unification of n_3 with $*$ is to hide the exported variable x from μ_3. By doing this, we block the creation of ε, and hence the unsafe unification.

A second approach is to keep x as an exported variable in μ_3, but modify the body of μ_2 by breaking the join between the first and the second position of R_1

$$\mathsf{R}_1(x, z, y) \wedge \mathsf{S}_1(y) \rightarrow \mathsf{T}_1(y) \qquad (\mu_2')$$

By doing this, we prevent the creation of ε, since the instance computed at line 2 of Algorithm 1 would consist of the facts $\mathsf{R}_1(\mathsf{n}_3, \mathsf{n}_5, *)$, $\mathsf{R}_1(*, *, \mathsf{n}_4)$, $\mathsf{S}_1(*)$ and, hence, there would be no homomorphism from $\text{body}(\mu_3)$ into it. Note that the modification of μ_2 to μ_2' is safe. Intuitively, this holds, since we break joins, and thus, we export less information.

Before presenting Algorithm 4, we will introduce some new notation. The *depth* of each bag β, denoted as $\text{depth}(\beta)$, coincides with the highest derivation depth of the facts in β. The *support* of a bag β, denoted as $\beta^{\prec}$, is inductively defined as follows: if $\text{depth}(\beta) = 1$, then $\beta^{\prec} = \beta$; otherwise, if $\text{depth}(\beta) > 1$, then $\cup_{\beta' \prec \beta} \beta'^{\prec}$. Consider an active trigger h for δ in I leading to the creation of a bag β. We use the following notation: $\text{dependency}(\beta) = \delta, \text{trigger}(\beta) = h$ and $\text{premise}(\beta) = h(\text{body}(\delta))$. Two bags β_1 and β_2 are candidates for modifyBody if $\beta_1 \prec \beta_2$, $\text{depth}(\beta_1) = 1$, $\text{depth}(\beta_2) = 2$ and there exists at least one repeated variable in the body of $\text{tgd}(\beta_1)$.

Algorithm 4 presents an iterative process for repairing a partially-safe Σ, by employing the three ideas we described above: checking for homomorphism violations within each bag and preventing unsafe unifications either by hiding exported variable, or by modifying the bodies of the s-t tgds. In brief, at each iteration $i \geq 0$, the algorithm repairs one or more dependencies from Σ_i, where $\Sigma_0 = \Sigma$, and incrementally computes the visible chase of the new

Algorithm 4 srepair$(\Sigma, \mathcal{V}, \text{prf}, n)$

1: $\Sigma_0 := \Sigma$
2: $B_0 := \text{visChase}_S(\Sigma)$
3: $i := 0$
4: **do**
5: $\quad \Sigma_{i+1} := \Sigma_i$
6: $\quad cont :=$ **false**
7: $\quad$ **if** $\exists$ unsafe $\beta \in B_i$, s.t. depth$(\beta) \leq$ depth(β'), $\forall$ unsafe bag $\beta' \in B_i$ **then**
8: $\quad\quad cont :=$ **true**
9: $\quad\quad$ **if** $i < n$ **then**
10: $\quad\quad\quad r_1 := \emptyset$; $r_2 :=$ hideExported$(\beta, \mathcal{V}, \text{prf})$
11: $\quad\quad\quad$ **if** $\exists \beta_1, \beta_2 \in \beta^{\prec}$, s.t. β_1, β_2 are candidates for modifyBody **then**
12: $\quad\quad\quad\quad r_1 :=$ modifyBody$(\text{tgd}(\beta_1), \text{tgd}(\beta_2), \text{prf})$
13: $\quad\quad\quad$ **if** $r_1 \neq \emptyset$ and it is preferred over r_2 w.r.t. prf **then**
14: $\quad\quad\quad\quad$ **remove** tgd(β_1) from Σ_{i+1}
15: $\quad\quad\quad\quad$ **add** r_1 to Σ_{i+1}
16: $\quad\quad\quad$ **else**
17: $\quad\quad\quad\quad$ **remove** tgd(β) from Σ_{i+1}
18: $\quad\quad\quad\quad$ **add** r_2 to Σ_{i+1}
19: $\quad\quad$ **else**
20: $\quad\quad\quad$ **if** $\nexists \beta'$, s.t., $\beta \prec \beta' \in B_i$ **then**
21: $\quad\quad\quad\quad$ **add** hideExported$(\beta, \mathcal{V}, \text{prf})$ to Σ_{i+1}
22: $\quad\quad\quad$ **else remove** tgd(β) from Σ_{i+1}
23: $\quad$ **compute** J_{i+1} from Σ_i, Σ_{i+1} and B_i
24: $\quad i = i + 1$
25: **while** *cont* **and** $i \leq n$
26: **return** Σ_n

set of dependencies, lines 4–25. Algorithm 4 terminates either when the dependencies are safe, or when the maximum number of iterations n is reached, line 25, in which case it repairs all unsafe dependencies by hiding their exported variables. The algorithm starts by initializing Σ_0 to Σ, lines 1. Then, at each iteration i, it first identifies the lowest depth unsafe bag, line 7, and attempts to repair the dependencies from Σ_i that lead to its creation, lines 7–22. If $i < n$, it proposes two different repairs for Σ_i, one based on hiding exported variables through hideExported (Algorithm 5), and the second based on eliminating joins through modifyBody (Algorithm 6), lines 10–19. Algorithm 4 applies the modifyBody if there exist two bags in the support of β that are candidates for modifyBody. Informally, Algorithm 4 tries to apply modifyBody as early as possible (condition depth$(\beta_1) = 1$, depth$(\beta_2) = 2$) and when there are one or more repeated variables in the body of tgd(β_1) (recall Example 5.5). Otherwise, if $i = n$, it either applies the function hideExported, or it eliminates the s-t tgds that are responsible for unsafe unifications.

Note that when we reach the maximum number of iterations we do not apply modifyBody. This is due to the fact that modifyBody might lead to unsafe unification of labeled nulls to $*$ that were not taking place before the modifying the s-t tgd through modifyBody. In contrast, hideExported is a safe modification, since it does not lead to new unsafe unifications.

Theorem 5.6. *For any partially-safe* $\mathcal{M} = (S, T, \Sigma)$, *any* $\mathcal{M}_V = (S, V, \mathcal{V})$, *any preference function* prf *and* $n \geq 0$, *Algorithm* srepair *returns a mapping* $\mathcal{M}' = (S, T, \Sigma')$ *that* preserves the privacy of $\mathcal{M}_V$ on all instances of S.

Proof. (Sketch) Since srepair takes as input a partially-safe mapping $\mathcal{M} = (S, T, \Sigma)$, it follows from Definition 5.1 that there

Algorithm 5 hideExported$(\beta, \mathcal{V}, \text{prf})$

1: $J :=$ premise(β)
2: $\nu := \emptyset$
3: **for each** $n \in$ Nulls occurring into J **do**
4: $\quad$ **add** $\{n \mapsto x\}$ to ν, where x is a fresh variable
5: $\mathcal{R} := \emptyset$
6: **for each** $\xi : \nu(J) \to I_S(\mathcal{V})$ **do**
7: $\quad \mu :=$ tgd(β)
8: $\quad$ **for each** $x \in$ dom(ξ) **do**
9: $\quad\quad$ **if** $\xi(x) \neq *$ **then**
10: $\quad\quad\quad$ **for each** $y \in$ exported(μ) **do**
11: $\quad\quad\quad\quad$ **if** $\tau(y) = \nu^{-1}(x)$, where $\tau =$ trigger(β) **then**
12: $\quad\quad\quad\quad\quad$ **remove** y from exported(μ)
13: $\quad$ **if** $\mu \neq$ tgd(β) **then**
14: $\quad\quad$ **add** μ to $\mathcal{R}$
15: **choose** the best repair μ_r of μ from $\mathcal{R}$ based on prf
16: **return** μ_r

Algorithm 6 modifyBody$(\mu_1, \mu_2, \text{prf})$

1: $\mathcal{R} := \emptyset$
2: **if** $\exists$ one or more repeated variables in body(μ_1) **then**
3: $\quad$ **for each** $\xi :$ body$(\mu_2) \to$ body(μ_1) mapping some $x_1 \in$ exported(μ_1) into some $x_2 \notin$ exported(μ_2) **do**
4: $\quad\quad$ Let $B \subseteq$ body(μ_1), s.t. $\xi($body$(\mu_2)) = B$
5: $\quad\quad$ Let V be the set of repeated variables from B
6: $\quad\quad$ Let P be the set of positions from B, where all variables from V occur
7: $\quad\quad$ **for each** non-empty $S \subset P$ **do**
8: $\quad\quad\quad \mu := \mu_1$
9: $\quad\quad\quad$ **replace** the variables in positions S of μ by fresh variables
10: $\quad\quad\quad$ **add** μ to $\mathcal{R}$
11: **choose** the best repair μ_r of μ from $\mathcal{R}$ based on prf
12: **return** μ_r

exists a homomorphism from chase$(\Sigma^{-1}, \text{Crt}_T) \setminus \text{Crt}_T$ into $I_S(\mathcal{V})$. Furthermore, from Proposition 5.2, we know that for each $\mu \in \Sigma$, there exists a homomorphism from body(μ) into $I_S(\mathcal{V})$ mapping each $x \in$ exported(μ) into the critical constant $*$. Due to the above, since the steps in lines 16–20 of Algorithm 1 do not introduce new labeled nulls and since srepair applies the procedure hideExported to each unsafe bag β in B_n, if there does not exist a bag $\beta' \in B_n$, such that $\beta \prec \beta'$, it follows that $\mathcal{M}'$ preserves the privacy of $\mathcal{M}_V$ on all instances of S, if hideExported prevents dangerous unifications of labeled nulls with the critical constant in line 4 of Algorithm 1. In particular, assume that we are in the n-th iteration of the steps in lines 4–25 of Algorithm 4. Let $\beta_n^0, \ldots, \beta_n^M$ be the unsafe bags in B_n. Assume also that for each $1 \leq l \leq M$, β_n^l, was derived due to some active trigger h^l, for some derived egd $\varepsilon^l \in \Sigma_{\approx}$ in I_j, where $j \geq 0$, line 17 of Algorithm 1. Let $\mu^l =$ tgd(ε^l), for each $0 \leq l \leq M$ and let μ_r^l be the repaired s-t tgd. Finally, let $\beta_{n+1}^0, \ldots, \beta_{n+1}^N$ be the bags in B_{n+1}, line 23 of Algorithm 4. Based on the above, in order to show that Theorem 5.6 holds, we need to show that (i) the number of bags in B_{n+1} is $\leq$ the number of bags in B_n and that (ii) the s-t tgds in $\left(\Sigma \setminus \bigcup_{l=0}^{M} \mu^l\right) \cup \bigcup_{l=0}^{M} \mu_r^l$ are safe. In order to show (i) and (ii), we consider the steps in Algorithm 5: for each $1 \leq l \leq M$, each exported variable y occurring in μ^l, which leads to an unsafe

	min	max	step
# s-t tgds per scenario (n_{dep})	100	300	50
# body atom per s-t tgds (n_{atoms})	1	3 (5)	–
# exported variables per s-t tgds (n_{vars})	5	8	–

Table 1: Properties of the generated iBench scenarios.

unification, line 11 of Algorithm 5, is turned into a non-exported variable. □

By combining Proposition 5.3 and Theorem 5.6 we can prove the correctness of Algorithm 2. Furthermore, if the preference function always prefers the repairs computed by hideExported from the repairs computed by modifyBody, we can show the following:

Proposition 5.7. *For each mapping* $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$*, each* $\mathcal{M}_\mathbf{V} = (\mathbf{S}, \mathbf{V}, \mathcal{V})$ *and each preference function* prf *that always prefers the repairs computed by* hideExported *from the repairs computed by* modifyBody*, Algorithm 2 returns a non-empty mapping that is safe w.r.t.* $\mathcal{M}_\mathbf{V}$*, if such a mapping exists.*

Proof. (Sketch) From Algorithm 3, we can see that frepair always computes a non-empty partially-safe mapping, if such a mapping exists. Note that a mapping, where no variable is exported and no repeated variables occur in the body of the s-t tgds is always partially-safe as long as, the predicates in the bodies of the s-t tgds are the same with the ones occurring in the policy views. Please also note that such a mapping is always considered by frepair. The above argument, along with the fact that a partially-safe mapping can be transformed into a safe one by turning exported variables into non-exported ones by means of the function hideExported, shows that Proposition 5.7 holds. □

6 EXPERIMENTS

We gauge the efficiency of our repairing algorithm on two types of preference function: a hardcoded one and a learning-based preference function.

We evaluated our algorithm using a set of 3.6K diverse mapping scenarios each of which consisting of a set of policy views and a set of s-t tgds. The characteristics of the scenarios are summarized in Table 1. In each scenario, we used a different number of s-t tgds n_{dep}, a different number of body atoms n_{atoms} and a different number of exported variables n_{vars}. The source schemas and the policy views have been synthetically generated using `iBench`, the state-of-the-art data integration benchmark [2]. We considered relations of up to five attributes and we created mappings using the `iBench` configuration recommended by the authors of [2]. We generated a set of varied policy views by applying the `iBench` operators copy, merge, deletion of attributes and self-join, each of which has been applied 10 times.

We implemented our algorithm in `Java` and we used the `Weka` library [16] that provides an off-the-shelf implementation of the k-NN algorithm for the learning-based preference function. We ran our experiments on a laptop with one 2.6GHz 2-core processor, 16Gb of RAM, running Debian 9.

In the remainder, all data points have been computed as an average on a total of 5 runs preceded by one discarded cold run.

Running time of repair**.** First, we study the impact of the number of s-t tgds and body atoms on the running time of repair. We adopt a fixed preference function that chooses the repair with the maximum number of exported variables. In case of ties, the repair with the maximum number of joins is preferred. We vary the number of s-t tgds from 100 to 300 by steps of 50 and the number of body atoms from 3 to 5, respectively. The obtained results are shown in Figure (3a), illustrating the fact that the median repairing time is less than 1.5s in most cases. For the most complex scenario containing up to five body atoms per s-t tgd, the median running time is less than 8s with 71s being the maximum. These results clearly show the high performance of our repair method along with its scalability. The reader should keep in mind that the repairing process is triggered prior to exchanging the data between source and target and might be rerun each time a mapping (set of s-t tgds) is modified or each time a policy view is modified, thus bringing the overhead to be quite reasonable in both cases.

prediction	golden standard μ_1	golden standard μ_2
μ_1	230	0
μ_2	0	395680

(a) P_{max} **confusion matrix.**

prediction	golden standard μ_1	golden standard μ_2
μ_1	290	1
μ_2	42	395577

(b) P_{avg} **confusion matrix.**

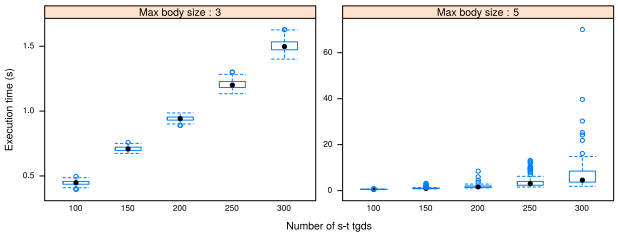
Table 2: Confusion matrix for the golden standards.

Figure (3b) shows the time breakdown for repair. The first bar shows the average running time to run the visible chase over the input s-t mappings, the second one shows the average running time for checking the safety of the computed bags and the third one shows the average running time for repairing the s-t tgds. The results show that the repairing time is $32x$ greater than the time to compute the visible chase and $40x$ greater than the time to check the safety of the chase bags for scenarios with 300 s-t tgds. In the simplest scenarios, these numbers are much lower (reduced to $5x$ and $9x$, respectively). Overall, the absolute values of the rewriting times are kept low (of the order of few seconds) for all these scenarios and gracefully scale while increasing the number of s-t tgds and the number of atoms in their bodies.

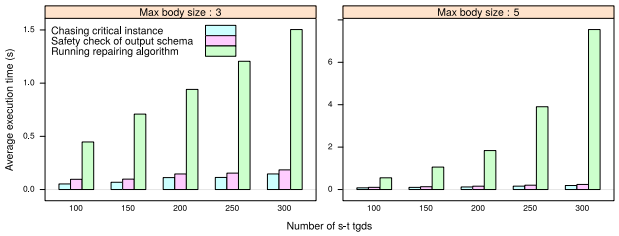
Time breakdown between frepair **and** srepair**.** Figure (3c) shows the average running time for frepair and srepair for the considered scenarios. We can see that srepair is the most time-consuming step of our algorithm. We can also see that the running time of srepair increases more in comparison to the running time of frepair when increasing the number of the s-t tgds and the number of atoms in their bodies. This is due to the incurred overhead during the incremental computation of the visible chase after repairing a s-t tgd (line 23 of Algorithm 4). Figure (3d) shows the correlation between the number of active triggers detected while incrementally computing the visible chase and the running time of srepair for scenarios with 100 s-t tgds using the ANOVA method ($p\text{-}value < 2.2e^{-16}$). Figure (3d) shows that the most complex scenarios lead to the detection of more than 45,000 active triggers. Despite the high number of the detected active triggers, the running time of srepair is kept low thus confirming its efficiency.

Leveraging learning-based preferences. We adopted the following steps in order to evaluate the performance of our learning approach. First, we defined the following two golden standard preference functions that we will try to learn:

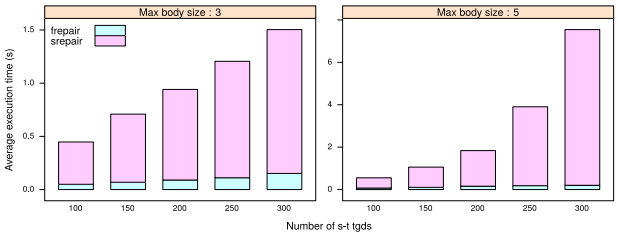
- P_{max}, which chooses the repair with the maximum number of exported variables and in case of ties, it chooses the repair with the maximum number of joins.
- P_{avg}, which computes the average number of exported variables and joins for each repair, and choose the one with the maximum average value.



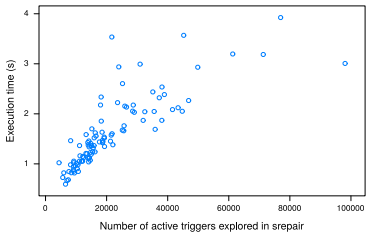
(a) Repairing times.



(b) Time comparisons.



(c) Time breakdown between frepair and srepair.



(d) Running time of srepair over 100 s-t tgds.

Figure 3: Summary of the performance-related experimental results.

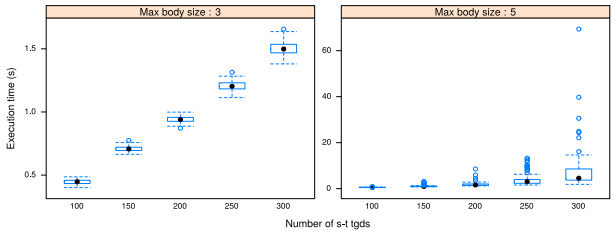


Figure 4: Repairing times with ML classifiers.

For both preference functions, we created a training set of 10,000 measurements for the k-NN classifier by running the repairing algorithm on fresh scenarios of 50 s-t tgds and five body atoms per s-t tgd. For each input vector $\langle \delta_{FV}, \delta_J \rangle$ whose repair we wanted to predict, we computed the Euclidean distance between $\langle \delta_{FV}, \delta_J \rangle$ and the vectors of the training set. We also set the value of parameter k to 1. This parameter controls the number of neighbors used to predict the output. Higher values of this parameter led to comparable predictions and are omitted for space reasons. Finally, we used the trained k-NN classifier as a preference function in srepair, rerun the above scenarios and compared the returned repairs with the ones returned when applying the golden standards P_{max} and P_{avg} as preference functions.

Learning P_{max}. Table (2a) (left) reports the confusion matrix associated to learning P_{max}, including the choices made by the k-NN classifier during its iterations.

Let us call μ_1 and μ_2 two possible repairs of an s-t tgd as evaluated by the k-NN classifier. We can observe that the prediction of μ_1 was correct (and equal to the golden standard in the training set) in 230 cases, while the prediction of μ_2 was correct in 395,680 cases.

This confirms the fact that μ_2 is the best repair across the iterations of the k-NN algorithm and is also chosen in case μ_1 and μ_2 are equally weighed by the preference function.

Furthermore, we also report the accuracy of learning the preference function, obtained by measuring the closeness of the learned mapping to the golden standard mapping.

We used the Matthews Correlation Coefficient metric (MCC) [3] to compare the repairs returned by the trained k-NN classifier and the ones returned when applied P_{max}. This is a classical measure that allows to evaluate the quality of ML classifiers when ranking is computed between two possible values (in our case, the choice between μ_1 and μ_2). This measure has been computed using the following formula:

$$MCC = \frac{N_{1,1} \times N_{2,2} - N_{1,2} \times N_{2,1}}{\sqrt{(N_{1,1} + N_{1,2})(N_{1,1} + N_{2,1})(N_{2,2} + N_{1,2})(N_{2,2} + N_{2,1})}}$$

where $N_{i,j}$ is the number of predictions of μ_i when μ_j is expected. The results of MCC range from -1 for the cases where the model perfectly predicts the inverse of the expected values, to 1 for the cases where the model predicts the expected values. The value $MCC = 0$ means that there is no correlation between the predicted value and the expected one. By applying MCC to the learning of P_{max}, we observed that the data are clearly discriminated, thus leading to high-quality of our prediction in this case ($MCC = 1$).

Learning P_{avg}. Table (2b) (right) shows the confusion matrix associated to learning P_{avg}. We can see that the predictions are less accurate in this case. The data is not as clearly discriminated as before, leading to a fairly negligible error rate ($< 0.02\%$). However, the latter is still acceptable for learning, since only $< 0.02\%$ of the predictions are erroneous. This is corroborated by an MCC value equal to 0.93, thus leading to a fairly acceptable quality of the prediction in this case too.

Running time of repair with ML classifiers. In the last experiment, we want to measure the impact of learning on the performance of our algorithm. To this end, we compare the running time of repair when adopting a hard-coded preference function (as in the results reported in Figure 3) and when adopting a learned preference function. Figure 4 shows the running times for the same scenarios used in Figure 3. We can easily observe that the runtimes are rather similar with and without learning and the difference amounts to a few milliseconds. This further corroborates the utility of learning the preference function and shows that the learning is robust and does not deteriorate the performances of our algorithm.

Qualitative study. In order to illustrate the utility of our approach, in this experiment we study possible rewritings of a mapping defined over the NHS schema.The NHS schema focuses on storing information concerning patients admitted in hospitals. Here, we consider the dependencies involving general information on patients. This includes administrative information

Relation	#atts
birth	34
patient	17
mothers_social_data	8
pis_e_prescribing	27
death	50

(a) Source schema characteristics

Relation	#atts
birth_export	34
patient_export	17
pis_e_prescribing_export	27
death_export	50

(b) Target schema characteristics

Relation	#atts
link_death_drugs	3
death_causes	20
prescribed_drugs_evolution	9
mothers_social	8
fathers_social	6

(c) Policy views schema characteristics

(1) $\text{birth}(\dots) \rightarrow \text{birth\_export}(\dots)$

(2) $\text{patient}(\dots) \rightarrow \text{patient\_export}(\dots)$

(3) $\text{pis\_e\_prescribing}(\dots) \rightarrow \text{pis\_e\_prescribing\_export}(\dots)$

(4) $\text{death}(\dots) \rightarrow \text{death\_export}(\dots)$

(d) Mapping over NHS

$\text{death}(\dots) \wedge \text{pis\_e\_prescribing}(\dots) \rightarrow \text{link\_death\_drugs\_data}(\dots)$

$\text{death}(\dots) \rightarrow \text{death\_causes}(\dots)$

$\text{pis\_e\_prescribing}(\dots) \rightarrow \text{prescribed\_drugs\_evolution}(\dots)$

$\text{birth}(\dots) \wedge \text{patient}(\dots) \rightarrow \text{mother\_social}(\dots)$

$\text{birth}(\dots) \rightarrow \text{fathers\_social}(\dots)$

(e) Policy views over NHS

Table 3: Properties of the NHS dataset.

Rewrited tgd	#possible repairs	#frontier variables in repairs	
		min	max
(1)	3	25	29
(2)	2	13	14
(3)	2	18	18
(4)	2	25	25

Table 4: Properties of the repairing process.

(relation patient in the source schema), social and medical information on the patient himself (relations birth and death), social data on patients' mothers (relation mother_social_data) and information on drugs prescriptions (relation pis_e_prescribing).

The characteristics of the source schema are summarized in Table 3a. The mapping to rewrite and the characteristics of its target schema are summarized in Tables 3d and 3b, respectively. The set of policy views and the characteristics of their target schema are reported in Tables 3e and 3c, respectively.

The link_death_drugs_data view allows to link the prescribed drugs with patient pathology, but no personal information is exported to prevent the identification of the patient. The death_causes view gives access to the causes of death of the admitted patients. The prescribed_drugs_evolution view gives access to drug prescriptions information without any identifying information. The mother_social and fathers_social views give access to patients' mothers and fathers social information.

In Table 4, we show the number of possible repairs for each tgd in Table 3d. It can be seen that the tgd (1) has three possible repairs, exporting from 25 to 29 variables, respectively. Analogously, the tgd (2) leads to two possible repairs, allowing to export from 13 to 14 variables each. Both tgds (3) and (4) lead to two possible repairs, with a constant number of exported variables for each tgd. These rewritings are distinguished by the exported variables, and one can decide to choose which repairing fits best her needs either visually or by leveraging the user preference function, as shown in our previous experiment.

7 CONCLUSION

We have studied the problem of data exchange in the presence of privacy restrictions expressed as policy views on the source schema. We have proposed a repairing process for the mappings that are unsafe under the source policy views. Our approach is inherently data-independent and leads to repairing the mappings guaranteeing privacy preservation at a schema level. As such, our approach is orthogonal to several data-dependent privacy-preservation methods (such as differential privacy methods), that can be used on the source and target instances to further corroborate the privacy guarantees. The study of such fruitful combinations of methods is devoted to future work.

We also envision several other extensions of our work, such as the study of more expressive GLAV mappings and the interplay between data-independent and data-dependent privacy methods as well as the usage of other learning methods.

ACKNOWLEDGEMENTS

Research by the first author is funded by ANR (under Grant No. 18-CE23-0002 QualiHealth)

REFERENCES

[1] Marcelo Arenas, Pablo Barceló, Leonid Libkin, and Filip Murlak. 2014. *Foundations of Data Exchange.* Cambridge University Press.
[2] Patricia C Arocena, Boris Glavic, Radu Ciucanu, and Renée J Miller. 2015. The iBench integration metadata generator. In *Proceedings of VLDB.*
[3] Pierre Baldi, Søren Brunak, Yves Chauvin, Claus AF Andersen, and Henrik Nielsen. 2000. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics* 16, 5 (2000).
[4] Zohra Bellahsene, Angela Bonifati, and Erhard Rahm (Eds.). 2011. *Schema Matching and Mapping.* Springer.
[5] M. Benedikt, B. Cuenca Grau, and E. Kostylev. 2017. Source Information Disclosure in Ontology-Based Data Integration.. In *AAAI.*
[6] Philip A. Bernstein. 2005. The many roles of meta data in data integration. In *In ACM SIGMOD.*
[7] Joachim Biskup and Piero Bonatti. 2004. Controlled query evaluation for enforcing confidentiality in complete information systems. *International Journal of Information Security* 3, 1 (2004).
[8] Joachim Biskup and Torben Weibert. 2008. Keeping secrets in incomplete databases. *International Journal of Information Security* 7, 3 (2008), 199–217.
[9] Piero A. Bonatti, Sarit Kraus, and VS Subrahmanian. 1995. Foundations of secure deductive databases. *IEEE TKDE* 7, 3 (1995), 406–422.
[10] Angela Bonifati, Ugo Comignani, and Efthymia Tsamoura. 2019. MapRepair: Mapping and Repairing under Policy Views (demo). In *ACM SIGMOD.* 1873–1876.
[11] Laura Chiticariu and Wang Chiew Tan. 2006. Debugging Schema Mappings with Routes. In *Proceedings of VLDB.* 79–90.
[12] Ugo Comignani. 2020. MapRepair - open source code. https://github.com/ucomignani/MapRepair.git.
[13] Remy Delanaux, Angela Bonifati, Marie-Christine Rousset, and Romuald Thion. 2018. Query-Based Linked Data Anonymization. In *ISWC 2018.* 530–546.
[14] Remy Delanaux, Angela Bonifati, Marie-Christine Rousset, and Romuald Thion. 2019. RDF Graph Anonymization Robust to Data Linkage. In *WISE 2019.* 491–506.
[15] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science* 9, 3-4 (2014), 211–407.
[16] Frank Eibe, MA Hall, and IH Witten. 2016. The WEKA Workbench. Online Appendix for" Data Mining: Practical Machine Learning Tools and Techniques. *Morgan Kaufmann* (2016).
[17] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. 2005. Data exchange: semantics and query answering. *Theoretical Computer Science* 336, 1 (2005).
[18] Bernardo Cuenca Grau, Evgeny Kharlamov, Egor V. Kostylev, and Dmitriy Zheleznyakov. 2015. Controlled Query Evaluation for Datalog and OWL 2 Profile Ontologies. In *IJCAI.*
[19] Bernardo Cuenca Grau and Egor V. Kostylev. 2016. Logical Foundations of Privacy-Preserving Publishing of Linked Data. In *AAAI.* 943–949.
[20] Gerome Miklau and Dan Suciu. 2007. A formal analysis of information disclosure in data exchange. *J. Comput. Syst. Sci.* 73, 3 (2007).
[21] Alan Nash and Alin Deutsch. 2007. Privacy in GLAV Information Integration. In *ICDT.*
[22] Muhammad I. Sarfraz, Mohamed Nabeel, Jianneng Cao, and Elisa Bertino. 2015. DBMask: Fine-Grained Access Control on Encrypted Relational Databases. In *In CODASPY.* 1–11.
[23] George L Sicherman, Wiebren De Jonge, and Reind P Van de Riet. 1983. Answering queries without revealing secrets. *ACM TODS* 8, 1 (1983), 41–59.
[24] Latanya Sweeney. 2002. k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 5 (2002), 557–570.

DomainNet: Homograph Detection for Data Lake Disambiguation

Aristotelis Leventidis Laura Di Rocco Wolfgang Gatterbauer
Renée J. Miller Mirek Riedewald
Northeastern University
Boston, MA, USA
{leventidis.a,la.dirocco,w.gatterbauer,miller,m.riedewald}@northeastern.edu

ABSTRACT

Modern data lakes are deeply heterogeneous in the vocabulary that is used to describe data. We study a problem of disambiguation in data lakes: *how can we determine if a data value occurring more than once in the lake has different meanings and is therefore a homograph?* While word and entity disambiguation have been well studied in computational linguistics, data management and data science, we show that data lakes provide a new opportunity for disambiguation of data values since they represent a massive network of interconnected values. We investigate to what extent this network can be used to disambiguate values.

`DomainNet` uses network-centrality measures on a bipartite graph whose nodes represent values and attributes to determine, without supervision, if a value is a homograph. A thorough experimental evaluation demonstrates that state-of-the-art techniques in domain discovery cannot be re-purposed to compete with our method. Specifically, using a domain discovery method to identify homographs has a precision and a recall of 38% versus 69% with our method on a synthetic benchmark. By applying a network-centrality measure to our graph representation, `DomainNet` achieves a good separation between homographs and data values with a unique meaning. On a real data lake our top-200 precision is 89%.

1 INTRODUCTION

Data lakes are large repositories where the metadata, including table names, attribute names, and attribute descriptions may be incomplete, ambiguous, or missing [32]. Modern data lakes are heterogeneous in many different ways: semantics, metadata, and data values. We consider the problem of determining if a data value (i.e., the value of an attribute in a table) that appears more than once in the data lake has a single meaning. A data value with more than one meaning is a *homograph*. We illustrate the data lake disambiguation problem through an example.

Example 1.1. *Consider the small sample of a data lake in Figure 1, showing four tables about different topics. T1 is about corporate sponsorship for efforts to save at-risk species, T2 is about populations in zoos, T3 is about car imports, and T4 is about corporate sales. Without disambiguation, a simple keyword search for Jaguar will return a very heterogeneous set of tuples.*

One approach to tackle this problem would be to apply document disambiguation by treating tables as documents. Such techniques are excellent at discerning topics in natural language documents and using this information to further disambiguate the words. However, because of the nature of tables that are often used to express relationships between different types of entities and values, distinguishing between a donor table T1 and a zoo table T2 that contain within them synonyms for animals while also being about very different topics (donations and zoos) is a difficult task. Distinguishing between car manufacturers T3 and corporations T4 can be even harder because of the prevalence of numerical values.

Entity resolution and disambiguation methods commonly assume a small set of tables about a small number of entity types (which may have the same or different schemas). In contrast, in a data lake the values to be disambiguated may appear in hundreds of tables about very different entity types and relationships between them. The ambiguous values need not be named entities, but may be descriptors or any data value in a table. This makes entity resolution inapplicable, but opens up new opportunities to use the large network of values and co-occurrences of values in the lake in new ways.

T1

Donor	At Risk	Donation
Google	Panda	1M
Volkswagen	Puma	2M
BMW	Jaguar	0.9M
Amazon	Pelican	1.5M

T2

name	locale	num
Panda	Memphis	2
Panda	Atlanta	2
Lemur	National	20
Jaguar	San Diego	8

T3

C1	C2	C3
XE	Jaguar	UK
Prius	Toyota	Japan
500	Fiat	Italy

T4

Name	Revenue	Total
Jaguar	25.80	43224
Puma	4.64	13000
Apple	456	370870
Toyota	123	123456

Figure 1: Running example with Jaguar and Puma having multiple meanings. How can we use co-occurrence information across a data lake to discern different meanings?

In entity resolution (ER) [9], the idea is to determine if two (or a set of) tuples refer to the same real-world entity or not. An important assumption in ER is that the tables being resolved are about the same (known) entity types. As an example, given a set of tables about papers that include authors as data values, we can determine if two tuples refer to the same paper (have the same meaning). As a by-product of entity resolution, a data value, for example "X. Wang," may be identified as an ambiguous data value that refers to more than one real-world entity. Schema-agnostic ER techniques have been proposed that do not assume the entities are represented by the same schema [37]. However, these approaches still assume the tables being resolved represent entities of the same type.

In our problem, we are not starting with a small set of tables that are known to refer to the same type of real-world entities, e.g., customers or research papers. We want to understand in a data lake with a massive number of tables if the value "Puma"

in T1 (see Figure 1), Attribute `At Risk` refers to the same real-world concept (not necessarily an entity) as "Puma" in Table T4, Attribute `Name`.

Disambiguation of words in documents has also been heavily studied [4, 24, 43, 49]. Solutions often rely on language structures or labeled training data. In contrast to documents, which are free text, tables are structured and lack the same intuitive notion of *context*. While plenty of research has explored disambiguation of documents, to the best of our knowledge there is no work on disambiguation of data lakes. This is of importance because data lakes can contain many data values that have different meanings. As an example, "Not Available" is a well known way to represent NULL values in a table. "Not Available" is not ambiguous from a natural-language point of view. However in a data lake it may appear in multiple attributes corresponding to names, telephone numbers, IDs etc., making "Not Available" a homograph meaning "unknown name" or "unknown number," etc.

Determining if a value in a data lake has a single or multiple meanings is unexplored territory. We define data lake disambiguation as follows:

Definition 1 (Data lake Disambiguation). *Given a data lake containing a collection of tables with possibly missing, incomplete, or heterogeneous table and attribute names. For any data value v that appears in more than one attribute (column) or table, determine if it has a single meaning or more than one meaning. The latter are called homographs.*

A homograph is not necessarily a single word from a dictionary or a vocabulary. In a data lake, a homograph can be a phrase, initialism (e.g., "NA"), identifier, or any blob (data value). We do not assume homographs to be named entities; they can be adjectives or another part of speech. Homographs arise naturally from words used in different contexts, e.g., the classic example of *Apple* as a fruit or a company, or *Jaguar* in Example 1.1. They can also arise due to errors, e.g., when animal color "yellow" is accidentally entered in the habitat column. We consider this now ambiguous value a homograph. Notice that updates to the data lake can change a homograph to a value with a single meaning, e.g., when the table with the only alternative meaning is removed; and vice versa.

In this work, we examine the global co-occurrence of data values within a data lake and how such information can be used to disambiguate data values. We show that a local measure is not sufficient and motivate why and how the full network of value co-occurrences enables effective disambiguation. This network exploits table structure and had not been considered in the most commonly studied disambiguation problems such as named-entity disambiguation and entity resolution. Its disambiguation power comes at a price: The value co-occurrence information is massive and it is not obvious how to process it efficiently for disambiguation.

Contributions. We address the data lake disambiguation problem using a network-based approach called `DomainNet`. Our main contributions are as follows.

- We define the problem of homograph detection in data lakes. Homographs may arise in tables that do not represent the same (or even similar) types of entities, and hence cannot be identified using entity resolution and disambiguation. They may not even be words in natural language and do not appear in natural-language contexts, making language models ineffective.
- We present `DomainNet`, a network-based approach to determine if a data value appearing in multiple attributes or tables is a homograph. `DomainNet` is motivated by work on community detection where a community represents a meaning for a value (e.g., animal or car model). A homograph is then a value that occurs in multiple communities. However, in the homograph detection problem (*i*) there are an *unknown* and possibly *large* number of meanings for a value and (*ii*) our goal is to find *values* that span communities, not the communities. We identify two measures for finding such community-spanning values, the *local clustering coefficient* [48] and the *betweenness centrality* [16], and empirically evaluate their usefulness in homograph detection.
- We present an evaluation on a synthetic dataset (with ground truth), studying the performance of both centrality measures and motivating the use of the more computationally expensive betweenness centrality. We compare `DomainNet` to a recent unsupervised domain detection algorithm D^4 [36] (any value belonging to multiple domains is a homograph). D^4 achieves a precision and a recall of 38% whereas `DomainNet` reaches 69%.
- We create a disambiguation benchmark from the real data used in a recent table-union benchmark [33] and show that we can effectively find naturally occurring homographs in this data (89% of the first 200 retrieved values are homographs based on ground truth). We also systematically introduce homographs into real data and show that betweenness centrality achieves 85% accuracy when homographs are injected into both small and large attributes, and over 97% accuracy when homographs are all injected into attributes with at least 500 distinct values. We show that `DomainNet` is effective even when there is high variance in the number of meanings of different homographs.
- To illustrate the importance of homograph discovery, we show the impact that as few as 50 homographs (injected into a clean unambiguous real data lake) can have on a domain discovery algorithm [36]. As the number of of homographs increases, the accuracy of the domain discovery algorithm deteriorates.
- The scalability of our approach depends on the size of the data lake vocabulary (the number of values) and on the density of the network (number of edges). We use real data (from NYC open data) with a vocabulary size of 1.5M to show that we can compute the `DomainNet` network in 3.5 min and find homographs in 27 min using an approximation of betweenness centrality based on sampling.

The remainder of this paper is organized as follows. In Section 2 we discuss existing work in disambiguation. In Section 3 we introduce our approach and describe how applying centrality measures on a graph representation of the data lake can be used to identify homographs. Section 4 summarizes the datasets used in our experimental evaluation presented in section 5. We conclude and outline possible future directions of our work in Section 6. For further information, please visit our project page at https://northeastern-datalab.github.io/table-as-query/

2 FOUNDATIONS OF DISAMBIGUATION

Disambiguation has been studied in several contexts in NLP, data management and broadly in AI and data science. We analyze how this work can be applied to disambiguation in data lakes.

2.1 Entity Resolution

Entity Resolution (ER) identifies records (also called tuples) across different datasets (or sometimes corpora) that represent the same

real-world entities. ER is generally applied to structured and semi-structured data including tables and RDF triples [18]. Some ER approaches also identify ambiguous values as part of the resolution process. For example, using collective entity resolution over two types of tables (e.g., papers and authors) one can identify if a value, say "X. Wang," refers to different authors [3]. Similarly in familial networks, one can resolve synonyms (different values that refer to the same person) and identify homographs (same value used to refer to different people) [26].

ER assumes that the information to be resolved or disambiguated is of a single known type (e.g., resolving customer tuples or patient records) or a small set of types (e.g., authors, their papers, and publishing venues). Some work, called schema-agnostic ER, does not require that all data be represented using the same schema [9]. However, all these approaches start with the assumption that two or more tables (or corpora) are describing the same type of entities [37, 38, 42].

In data lake disambiguation, we seek to find ambiguous values even when we do not know what type of entities a table is describing. We also do not know if different tables are describing the same or different entities. Hence, we cannot apply collective models or other resolution models that rely on this knowledge.

Example 2.1. *Given the four tuples with* `Jaguar`*:* `[BMW, Jaguar, 0.9M]`, `[Jaguar, San Diego, 8]`, `[XE, Jaguar, UK]`, *and* `[Jaguar, 25.8, 43224]`, *does* `Jaguar` *have the same meaning? These four tuples correspond to four different types of facts: donors and the amount they contribute to protect an endangered species, animals in zoos, car models, and economic information about companies. ER schema-agnostic algorithms are insufficient in resolving (or disambiguating) values within these heterogeneous tables because they rely on the hypothesis that the tables they examine refer to the same type of real-world entity.*

2.2 Semantic Type Detection

A possible approach to data lake disambiguation is to discover semantic types for all attributes (columns) and then label a value appearing in different semantic types a homograph. In the running example, identifying the semantic type of `T1.At Risk` and `T2.name` as animal and mammal, respectively, and knowing that mammals are animals, one can infer that Jaguar is not a homograph there. In contrast, recognizing `T3.C2` is of type "Car Manufacturer," which is neither a sub- nor super-type of animals, implies that Jaguar in `T3` and `T1` represents a homograph. Here, we discuss different approaches to semantic type discovery and to what extent they could be used for homograph detection.

Knowledge-based Techniques. There has been considerable work on semantic type detection in the Semantic Web community that uses external knowledge from well-known ontologies including DBpedia [28], Yago [46] and Freebase [5]. Most solutions have been applied to Web tables [11, 12, 29] that are small (in comparison to other data lakes) and have rich metadata (table and attribute names).

Hassanzadeh et al. [20] use a map-reduce approach to find similarity between a (column, data value) pair from a table with a (class, instance label) pair from the Knowledge Base (KB). Ritze et al. [41] match Web tables to DBpedia to profile the potential of Web tables for augmenting knowledge bases with missing information. These approaches cannot infer type information for an attribute that it is not part of the KB. Unfortunately, the coverage of values from data lakes in Open KBs is low (a recent study reports about 13% [33]), limiting their applicability.

Supervised Techniques. An alternative are machine learning (ML) techniques that infer the semantic type of attributes. ML solutions utilize a variety of graphical models (Conditional Random Fields [19], Markov Random Fields [31]), as well as Multi-level Classification [47], and Deep Learning [23]. Sherlock [23] uses features about the values in an attribute to classify some of the attributes in a data lake into one of 78 semantic types (like address or horse jockey) [23]. A recent solution, called SATO [51], augments this approach and shows that using row information can improve the classification accuracy for the same 78 semantic types. These approaches require large amounts of labeled training data and are limited by the set of pre-defined types.

Unsupervised Techniques. Unsupervised semantic type discovery algorithms have only recently started to be studied. We discuss two unsupervised algorithms, one for semantic type discovery, D^4 [36], and one for table unionability search [33].

D^4 provides an unsupervised approach with a focus on assembling all the values of each semantic type in a data lake [36] (these values are called a "domain"). They propose a data-driven approach that leverages value co-occurrence information to cluster values that are from the same domain. Heuristics attempt to deal with ambiguous values that may appear in multiple domains. In our context, D^4 can be used to label values that appear in multiple domains as homographs. This indeed serves as a baseline in our experiments.

Table Union Search [33] solves a different problem. Given a query table, they find a set of tables from the lake that are most unionable with it. In order to do so, they provide several similarity measures that are used collectively to calculate how unionable two attributes are. This work can use both ontological and semantic (word embedding) signals when present to determine unionability over heterogeneous attributes, but does not attempt to find or label homographs.

2.3 Disambiguation in Related Areas

Word-sense disambiguation (WSD) [24, 34], i.e., the task of identifying which meaning of a word is used in a sentence, is an important problem in computational linguistics. Although a human can proficiently perform this task on a document, constructing algorithms that perform this task effectively is still an open research problem. Techniques proposed so far range from dictionary-based methods, which use the knowledge encoded in lexical resources (e.g., WordNet) [34], to more recent solutions in which a classifier is trained for each distinct word on a corpus of manually sense-annotated examples [39]. Additionally, completely unsupervised methods have also been proposed that cluster occurrences of words, thereby inducing word senses, i.e, word embeddings [24]. The aforementioned solutions rely on information (or latent information) about the structure of sentences including grammatical rules. Finally, while solutions that do not rely on grammar also exist, they only operate on documents and not tables [4, 43].

Another relevant sub-task in Natural Language Processing is Named-Entity Recognition (NER), which has been proposed as a possible solution for disambiguation [49]. NER seeks to locate and classify named entities mentioned in unstructured text into predefined categories such as person names, organizations, locations, etc. NER systems have been created that use linguistic grammar-based techniques as well as statistical models [1].

A special case of the NER problem is the author name disambiguation problem [14, 44] Authors of scholarly documents often share names which makes it hard to distinguish each author's

work. Hence, author name disambiguation aims to find all publications that belong to a given author and distinguish them from publications of other authors who share the same name. Different solutions have been proposed using graphs [30]. However, the graph structure proposed is largely domain specific. The graph contains not only the information about the co-authorship and published papers, but also venue of the paper published, year of research activities and so on.

3 DISAMBIGUATION USING DOMAINNET

We now present our proposed solution, DomainNet[1], for finding homographs in a data lake.

3.1 Problem Definition

Recall from Definition 1 that a *homograph* is a data value that appears in at least two attributes with more than one meaning. Values that are not homographs are *unambiguous values*. In data lakes, attribute and table names can be missing or misleading (with many ambiguous terms like "name," "column 2," or "detail") [32]. Well-curated enterprise lakes may have more complete metadata, but even they do not follow the unique name assumption—which states that different attribute names always refer to different things. As a result, many data lake search approaches rely solely on the table contents [10, 13, 52, and others]. In a similar vein, in DomainNet, we investigate to what extent data values and the co-occurrence of data values within attributes can be used to determine if a value is a homograph.

Example 3.1. *In Figure 1, the data value* Jaguar *is a homograph because it refers to the animal in Tables T1 and T2 and refers to the car manufacturer in Tables T3 and T4. Other values such as* Panda *and* Toyota *are unambiguous since they only have a single meaning across* all *tables.* Puma *is also a homograph, appearing as an animal and a company. Figure 2 displays which values co-occur with* Jaguar *in the same column using an incidence matrix: the vertical axis shows the different values, and the horizontal axis the different attributes occurring in the data lake.*

Note that homographs need not be values from a dictionary. They can be any data value that appears in a table. Another example of a homograph is the data value 01223 *which in some attributes may refer to a Massachusetts zip code and in others to an area code near Cambridge, UK, and in yet others to the suffix of an Oil Filter Element Replacement product code.*

	Fiat	Toyota	Apple	Puma	Jaguar	Pelican	Panda	Lemur
T2.name					1		1	1
T1.At Risk				1	1	1	1	
T4.Name		1	1	1	1			
T3.C2	1	1			1			

Figure 2: Incidence matrix: vertical axis attributes, horizontal axis data values.

In a well-curated database or warehouse, we may know the semantic meaning of each attribute (e.g., "Animal Name" vs. "Company Name") and can leverage it to identify homographs. However, in a dynamic, non-curated data lake, we cannot rely on this information to be available.

[1]The code for DomainNet and our benchmarks is available at https://github.com/northeastern-datalab/domain_net.

3.2 DomainNet: Viewing Values as a Network

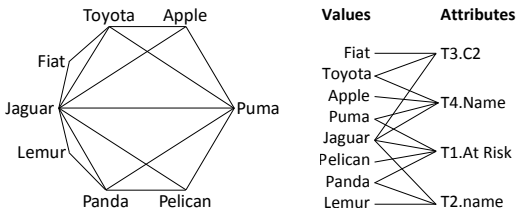
In data lakes, without *a priori* knowledge of table semantics or types, we take a network-based approach to understanding the meaning of repeated data values. We propose to detect homographs using network measures. For that purpose, we can interpret the co-occurrence information about values across different attributes using a network representation in which nodes represent data values and edges represent the fact that two values co-occur in at least one column (attribute) in the data lake.

Example 3.2. *In Figure 3, we depict the values from the same four attributes shown in Example 3.1. Figure 3a shows the value co-occurrence network. Notice that by removing both "Puma" and "Jaguar" the remaining nodes become* disconnected *into two components. This captures the intuition that those two values are pivotal in that they bridge two otherwise disconnected meanings or graph components.*

Whereas this representation allows us to apply straightforward metrics from community detection, it comes at a high cost: the representation uses more space than the original data lake. The incidence matrix is sparse and has as many entries as there are cells in the data lake (Figure 2). In contrast, the co-occurrence graph increases quadratically in size with respect to the cardinality of attributes (the size of the vocabulary) in the data lake (Figure 3a). Consider a single column with 100 values. The incidence matrix represents this information with 100 rows, 1 column, and 100 entries. The co-occurrence graph represents this with 100*99/2=4950 edges across 100 nodes.

Thus, we use a more compact network representation that allows us (after some modifications) to apply network metrics to discover pivotal points (Figure 3b). DomainNet uses a bipartite graph composed of (data) value nodes and attribute nodes. The attribute nodes represent the set of attributes and the value nodes the set of data values across all attributes in the lake. Every data value is treated as a single string, it is capitalized and has its leading and trailing white-space removed to ensure consistent comparison of data values across the lake. Notice that each data value, even if found in multiple attributes, is represented by one single value node in the graph. An edge is placed between a value node and an attribute node if the data value appears in the attribute (column) corresponding to that attribute node. Data values that appear in more than one attribute are candidates for being homographs.

Example 3.3. *Figure 3b, shows a portion of the* DomainNet *representation for Figure 1 using only the four attributes of Example 3.1.*



(a) Co-occurrence graph **(b) Bipartite graph**

Figure 3: Two graph representations of a portion of Figure 1.

In the DomainNet bipartite graph, we call two data values *neighbors* if they both appear in the same attribute (and hence there is a path of length two between them in the graph). Similarly,

two attributes are *neighbors* if they have at least one data value in common (and hence there is a path of length two between them). For a data value node v, $N(v)$ denotes the set of all its value neighbors. We also define the *cardinality of a data value* node v as the number of neighbors $|N(v)|$, which is the number of unique data values that co-occur with v. If n is the number of value nodes and a the number of attribute nodes, the number of edges in a DomainNet graph over real data tends to be much less than $n \cdot a$.

Tables to Graph. Recent work on embedding algorithms in relational databases [2, 7, 27] use a graph representation of tables. Like DomainNet, they model values and columns as nodes. Depending on the problem addressed, some approaches also include nodes for rows and tables. Like in our approach, column names are not assumed to be present or unambiguous.

Koutras et al. [27] and Capuzzo et al. [7] use a tripartite graph representation in which every value node is connected with its column node and its row node. Such an approach works well for the tasks of tuple-level entity resolution and for schema matching (a similar task to semantic type discovery). We experimented using both row and table information in DomainNet and found it was not useful in disambiguating values. In our example, Panda in $T1$ and $T2$ are not homographs, but the row information makes them seem quite different and we did not find it helpful.

In contrast, Arora and Bedathur [2] use a homogeneous graph using only data value nodes that are connected with each other if they appear in the same row of the table. They do not use the value co-occurrence information within a column, making homograph detection using solely row context inappropriate in large heterogeneous datasets.

3.3 Homograph-Disambiguation Methodology

Intuitively, data values that frequently co-occur with each other will form a latent semantic type or community in DomainNet, with many paths of varying length between them. Homographs will span two or more communities. Notice however that we do not know *a priori* what the communities are or even how many there are. While there is a rich literature on community detection, many approaches require knowledge of the possible communities such as the number of communities [8]. Others are parameter-free, meaning they can learn the number of communities [21, and others]. However, in our problem the number is not only unknown, it may be massive. A data lake with just a modest number of tables may have many attributes representing a multitude of different semantic types (communities of values) [8, 15].

What we propose in this paper is to use network centrality measures that can be defined without prior knowledge of how many communities exist, their overlap, or the distribution of attribute cardinalities. The intuition behind centrality measures is to capture how well connected the neighbors of a given node are. We define variants of these measures appropriate for the DomainNet bipartite graph. We then discuss to what extent these measures may distinguish whether a data value has a single meaning or multiple meanings (the latter being a homograph).

Local Clustering Coefficient as a homograph score. The local clustering coefficient (LCC) [48] for a given value node measures the average probability that a pair of the node's neighbors are also neighbors with each other, i.e., the fraction of value-neighbor triangles that actually exist over all possible triangles.

The LCC metric is usually defined over unipartite graphs (such as the co-occurrence graph in Figure 3(a)). We use the definition of value-neighbors (recall the set of all value neighbors of a value node u is $N(u)$) to generalize LCC to our bipartite graph.

The pairwise clustering coefficient of two data value nodes v and w is defined as the Jaccard similarity between their neighbors

$$c_{vw} = \frac{N(v) \cap N(w)}{N(v) \cup N(w)}.$$

Given a graph G and a value node u, the LCC is defined as the average pairwise clustering coefficient among all the node's value neighbors:

$$c_u = \frac{\sum_{v \in N(u)} c_{vu}}{|N(u)|}. \quad (1)$$

The LCC of a node u can be computed in time $O(N(u)^2)$ and provides a notion of the importance of a node in connecting different communities.

Hypothesis 3.4 (Homographs using LCC). *A value node corresponding to a value that is a homograph will have a lower local clustering coefficient than a value node with a single meaning.*

Intuitively, we expect unambiguous values to appear with a set of values that co-occur often and thus have high LCC scores. This behavior should be less common for homographs, which may span values from different communities as they appear in various contexts depending on their meaning.

Despite LCC's computational simplicity, the measure as defined in Equation (1) is no more than the average Jaccard similarity between the set of attributes that a value co-occurs with. Unfortunately, it is well-known that *Jaccard similarity is biased to small sets.* As consequence, *the measure is not as effective in real data lakes* where attribute sizes are often considerably skewed. Our experiments will confirm this downside of LCC.

Betweenness Centrality as a homograph score. The LCC of a node is fast to compute, but it only considers the local neighborhood of a value. In a data lake, the local neighborhood may not be sufficient. In particular, the local neighborhood may not include values that are members of the same community but happen to not co-occur. In order to overcome these two problems (missing values in the neighborhood and attributes with very different cardinalities) we look at metrics that take a more global perspective on the network.

The *betweenness centrality* (BC) of a node measures how often a node lies on paths between *all other nodes* (not just the neighbors) in the graph [16]. One way to think of this measure is in a communication network setting where the nodes with highest betweenness are also the ones whose removal from the network will most disrupt communications between other nodes in the sense that they lie on the largest number of paths [35].

Consider two nodes v and w. Let σ_{vw} be the total number of shortest paths between v to w, and let $\sigma_{vw}(u)$ be the number of shortest paths between v to w that pass through u (where u can be any node).[2] The betweenness centrality of a node u is defined as follows, where v and w can be any node in the graph:

$$BC(u) = \sum_{v \neq u, w \neq u} \frac{\sigma_{vw}(u)}{\sigma_{vw}}. \quad (2)$$

[2] Since the bipartite graph used in DomainNet is not homogeneous we also examined other variations of BC such as considering only values nodes as end points for the examined shortest paths. We found that using all nodes in the BC definition provided empirically the best results for finding homographs.

By convention $\frac{\sigma_{vw}(u)}{\sigma_{vw}} = 0$ if σ_{vw} (and therefore $\sigma_{vw}(u)$) is 0.

Intuitively, a homograph appears with sets of values that do not or rarely co-occur across those sets, and thus the shortest paths between such non-co-occurring nodes would have to go through the homograph node. Conversely, unambiguous values appear with a set of values that also co-occur a lot, and thus the shortest path between them does not unnecessarily have to go through one or a few nodes.

Hypothesis 3.5 (Homographs using BC). *A value node corresponding to a homograph will have a higher betweenness centrality than a value node with a single meaning.*

Example 3.6. *The LCC scores of the* Jaguar *and* Puma *data value nodes in Figure 1 are 0.36 and 0.43 respectively. The LCC scores of the other data value nodes that appear more than once,* Toyota *and* Panda*, are somewhat higher at 0.46. The BC scores of the* Jaguar *and* Puma *value nodes in Figure 1 are 0.025, 0.003 respectively. The BC of the other value nodes that appear more than once,* Toyota *and* Panda*, are at 0.002. Since this example only uses four small tables it does not expose the possibly different rankings between LCC and BC scores but suggests that BC, even on small graphs is more discerning.*

Complexity of BC. Calculating the BC for all nodes in a graph is an expensive computation. A naive implementation takes $O(n^3)$ time and $O(n^2)$ space (n denotes the number of nodes in the graph). The most efficient algorithm to date is Brandes' algorithm [6] that takes $O(nm)$ time and $O(n+m)$ space (for unweighted networks) where m is the number of edges in the graph. Notice that this algorithm is still expensive if the graph is dense (i.e., $m >> n$).

The high time complexity of BC motivated approximations, which usually sample a subset of nodes from the graph and thus do not calculate all shortest paths. One common sampling strategy is to pick nodes with a probability that is proportional to their degree (nodes with high degree are more likely to appear in shortest paths). Riondato and Kornaropoulos [40] provide an approximation algorithm via sampling with offset guarantees. Geisberger, Sanders, and Schultes [17] provide an approximation algorithm without guarantees that performs very well in practice. The complexity of the approximate BC is $O(sm)$ where s is the number of nodes sampled. We chose Geisberger, Sanders, and Schultes [17] to approximate betweenness centrality to benefit most from its short run-time on large graphs.

3.4 Disambiguation Using DomainNet

In this section, we describe the implementation of an end-to-end system which allows users to disambiguate data lakes using our proposed methodology. Our system has three steps as illustrated in Figure 4: (1) construct DomainNet graph; (2) calculate measures; and (3) rank measures.

DomainNet *graph construction.* The input is a set of raw data tables from relational databases, CSV files, or any other open data format. It is important to note that we do not require any information in regards to types, attribute names, or the semantics of relationships between tables. We build our bipartite graph as described in Section 3.2.

Graph measure computation. Using the DomainNet graph constructed in the previous step, our system computes both LCC and BC scores for each value node (Section 3.3). We show empirically in Section 5.1 that BC outperforms LCC in homograph detection.

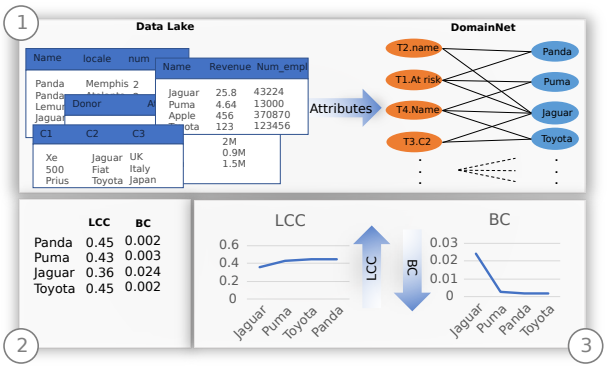


Figure 4: Disambiguation system on DomainNet. (1) Construct a DomainNet graph from a data lake. (2) Calculate BC and LCC scores for each value node in the graph. (3) Rank the scores accordingly.

Graph measure ranking. Nodes are ranked by their centrality score (ascending order for LCC measures, and descending order for BC measures) the top ranked data values present to a user.

4 DATASET DESCRIPTION

Homograph detection in data lakes is a new problem and no benchmarks are available for it. While many data lakes exist, they do not contain labels that identify the homographs. In addition to being a hugely expensive task when done manually, homograph labeling is not a one-time effort: when the content of the data lake changes, an unambiguous value can become a homograph or vice versa. Hence, benchmark design in this context constitutes a non-trivial contribution in itself.

We introduce the four datasets used for the evaluation of DomainNet. The first is a new synthetic benchmark and the other three contain real data. The second is an adaptation of the Table Union Search (TUS) Benchmark [33] that uses real tables from UK and Canadian open-data portals and which we adapt for our problem. The third is a modified version of TUS, called TUS-I, where we systematically inject homographs. The fourth, used to evaluate scalability, is a real data set from NYC Education Open Data, which was also used to evaluate a domain discovery approach [36].

Table 1 summarizes detailed statistics about the datasets. For each, we list the number of tables, the total number of attributes (columns) across all tables, the number of unique values in the data lake, the total number of homographs, the range of cardinalities of any homograph[3] (Card(H)), and the range of the number of distinct meanings, #M, (based on ground truth) the different homographs have across the data lake. All datasets can be found at https://github.com/northeastern-datalab/DomainNet-Datasets

Table 1: Four datasets and their statistics.

	#Tables	#Attr	#Val	#Hom	Card(H)	#M
SB	13	39	17,633	55	151-1,966	2
TUS - I	1,253	5020	163,860	N/A	N/A	N/A
TUS	1,327	9859	190,399	26,035	3-22,703	2-100
NYC-EDU	201	3496	1,469,547	N/A	N/A	N/A

[3]Recall the definition of the cardinality of a homograph node v as $|N(v)|$, which is the number of unique data values that v co-occurs with.

4.1 Synthetic Benchmark (SB)

We designed a small fully synthetic, but real-world inspired, data lake for a systematic validation of our approach. It consists of 13 tables generated using Mockaroo[4], which lets the data creator specify data sources from various categories.

Each table has 1000 rows, except for two tables that contain countries and states. We used the real numbers of countries and US states of 193 and 50, respectively. There are 55 data values that are homographs, e.g., Sydney (city or name), Jamaica (city or country), Lincoln (car or city), CA (country or state abbreviation), and Pumpkin (grocery product or movie title). The benchmark along with its metadata (full list of tables and their schemas and stats) are in our github.

4.2 Table Union Search Benchmark (TUS)

In the absence of homograph-labeled large real data lakes, we set out to find a closely related benchmark that we could adapt to our purposes. Unfortunately, while there are many table-based benchmarks, even those for data-semantics-related problems generally proved hard to adapt. For example, the VizNet corpus [22] used in semantic type detection in tables [23, 51] provided ground-truth labels for only a small fraction of the columns in the repository, making ground-truth discovery of all homograph labels practically impossible. We therefore selected the Table Union Search (TUS) benchmark [33], which contains real data and provides a ground-truth mapping for *each* column to the set of columns in the repository that it is unionable with. This enables us to automatically label all homographs. Let $U(a)$ denote the set of columns (attributes) a given column a is unionable with and notice that a is always unionable with itself, hence $a \in U(a)$. Let $A(n)$ be the set of columns (attributes) a data value n appears in. Converting the TUS benchmark into our bipartite graph representation, we can automatically label data values as "unambiguous" or "homograph" based on the unionability ground truth.

Definition 2 (Homograph in the table union search benchmark). *A data value n is a homograph if there exist two attributes a and a' in $A(n)$ such that $U(a) \neq U(a')$; otherwise n is an unambiguous value.*

Intuitively, a data value is a homograph if it appears in at least two different columns that are not unionable (and hence have different types). For instance, assume value `USA` appears in columns `country_x1` and `location_x2` in tables X1 and X2, respectively. If the corresponding two columns are unionable, i.e., $U(\texttt{country\_x1}) = U(\texttt{location\_x2}) = \{\texttt{country\_x1}, \texttt{location\_x2}\}$, then we can conclude that `USA` is an unambiguous value. In contrast, the columns containing the value `jaguar` in the zoo or donor tables are not unionable with either the company or car model tables and hence `jaguar` would be labeled a homograph.

Based on Definition 2 there are 164,364 unambiguous values and 26,035 homographs in the TUS benchmark, suggesting homographs are very abundant in real data lakes. Notice that attribute cardinalities in TUS have high skew, a common phenomenon in data lakes for open-data repositories [32]. Hence, this benchmark provides a "stress-test" for our approach. How well can it deal with both small and large cardinalities of attributes containing a homograph (in TUS these cardinalities range from 3 to 22,703).

[4] https://www.mockaroo.com/

4.3 TUS with Injected Homographs (TUS-I)

Having real data is important, but we also need to understand the performance of our solution as the number of homographs in a data lake changes. To this end, we modified the TUS benchmark as follows. First, we removed all 26,035 homographs. Second, we carefully introduce artificial homographs with different properties. Since the artificial homographs are now the only ones in the data lake, we can measure how their properties affect the detection algorithm.

A homograph is injected by selecting two different data values from two columns that are not unionable. These original values are then replaced by a new unique value such as "InjectedHomograph1". We only replaced string values with at least 3 characters. In our experiments, we vary the minimum allowed cardinality of the attributes containing values replaced with an injected homograph. We also vary the number of meanings of an injected homograph. This allows us to evaluate the effectiveness of our approach in identifying homographs with respect to the cardinality and number of meanings of the homographs.

5 EXPERIMENTAL EVALUATION

The main goal of the experiments is to evaluate how well `DomainNet` performs in terms of precision and recall for identifying the homographs in the benchmark datasets. We are particularly interested in determining if the more expensive betweenness centrality (BC) provides significant improvement over local clustering coefficients (LCC) (Section 3). Since a homograph candidate must appear in at least two different table columns, `DomainNet` pre-processes the input to remove data values that appear only once in the data lake. As a result, the corresponding graph representation has about 3% fewer nodes in the TUS benchmark and 30% fewer nodes in SB. Moreover we examine how our method scales with larger input graphs and how homographs can impact existing data integration tasks such as domain discovery.

Comparison to a baseline. There is no previous work that directly explores homograph detection in data lakes (Section 2), and previous work on the related problem of semantic type detection and domain discovery is generally supervised, i.e., requires labeled training data. Hence, the only suitable algorithm that we could reasonably adapt to solve our problem is the recently proposed state-of-art unsupervised domain-discovery algorithm D^4 [36]. We used the original code provided by the authors[5] with its default parameter settings. When applied to a data lake, D^4 assigns attributes to the discovered domains. A natural way to identify homographs then is to identify data values that appear in more than one of those domains. We compare D^4 to `DomainNet` on the synthetic benchmark as it only contains string values. D^4 discovers domains only for string data, making it ineffective on the TUS benchmark, which contains real data with many numerical attributes.

Measures of success. We generally measure precision and recall, which are reported for the k top-ranked homograph candidates identified by each of the algorithms. By default k is set to the true number of homographs in the data lake.

Software implementation. We implemented `DomainNet` in Python 3.8, using Networkit [6] [45] to calculate exact and approximate BC scores over our bipartite graph. This is a Python library for large-scale graph analysis whose algorithms are written in

[5] The code is available at https://github.com/VIDA-NYU/domain-discovery-d4.
[6] https://networkit.github.io

C++ and support parallelism. All our experiments were run on a commodity laptop with 16GB RAM and an Intel i7-8650U CPU.

5.1 Fully Synthetic Benchmark (SB)

We first use the SB to compare the homograph rankings obtained using the LCC and BC measures (Section 3) in order to study their ability to identify homographs. The bipartite graph for SB is relatively small, consisting of 17,672 nodes (17,633 data-value nodes and 39 attribute nodes) and 19,473 edges. We calculated the local clustering coefficients (LCC) and betweenness centrality (BC) for each node in the graph and examined how these scores differ between homographs and unambiguous values.

Which measure is better at discovering homographs? Figure 5 shows the top-55 data values based on LCC. For LCC, lower scores should in theory indicate a greater probability of being a homograph. Notice how more than 75% of the top-ranked data values are not homographs, meaning that a large number of unambiguous values have smaller LCC scores than the homographs. This is mainly caused by unambiguous values from small domains that do not co-occur often with many values in their domain. This confirms our hypothesis from Section 3 that LCC may not work well when homographs appear in small domains. In fact, the majority of the 55 homographs in the dataset have LCC scores significantly above 0.45 and so it is not necessarily true that homographs have low LCC cores. Overall the results indicate that LCC scores do not provide an effective separation between homographs and unambiguous values.

On the other hand, the BC scores result in a vastly better top-55 result as shown in Figure 6. Here 38 out of the top-55 BC scores correspond to homographs. This is a much improved outcome over the LCC scores in Figure 5. But what happened to the remaining 17 homographs that are not in the top-55? We noticed that the remaining 17 homographs have betweenness scores of nearly zero and they all are values corresponding to homographs that are abbreviations of country and state names. Recall that these are the only two tables in SB with fewer than 1000 tuples, where the state table contains only 50 tuples. This means that the BC score for values in these small domains cannot be very large as there cannot be as many shortest paths that would pass through the homograph in question.

An explanation for the low BC scores for these homographs is the fact that there is considerable intersection between the country and state values which is not the case with other homographs (e.g., the car brands and cities intersect only on the value Lincoln and Jaguar). This relatively large intersection also reduces the BC scores for those homographs as the number of shortest paths connecting two nodes between cities and states is much larger. For example, going from the country code GR to the state code MA, the shortest path could be using the homograph AL (which is for Albania/Alabama) or CA (which is for Canada/California) or any other homograph between countries and states. As a result those homographs receive lower BC scores, because the denominator in Equation (2) becomes large.

How good is previous work at finding homographs? As discussed earlier, we compare `DomainNet` against a competitor based on D^4[36]. When applied to the SB dataset, D^4 discovers four domains corresponding to `Country`, `Country Code`, `Scientific Animal Name`, and `Scientific Plant Name`. It maps the domains on 14 out of 39 table columns (attributes) in SB. Among these 14 attributes, there are 21 of the 55 homographs. Overall, when considering the top-55 results returned, the D^4-based algorithm disambiguates homographs in SB with a precision, recall, and F1-score of 38%. Using the BC score, `DomainNet` achieves for the top-55 results a precision, recall, and F1-score of 69%.

5.2 Experimental Evaluation on TUS-I

We now study the BC-score-based version of `DomainNet` in more detail on the large real-world dataset TUS-I with the injected homographs. Due to the cost of running BC for each node, all BC scores are approximated using 5000 samples. [7]

Table 2: % of the 50 injected homographs appearing in the top-50 results vs. cardinality of the data values replaced by the injected homograph. (Numbers are averages of 4 runs for each threshold.)

Cardinality of replaced values	> 0	≥ 100	≥ 200	≥ 300	≥ 400	≥ 500
% of injected homographs in top 50	85%	93.5%	93.5%	95%	94.5%	97.5%

How does cardinality affect homograph discovery? Recall that after removing all original homographs in TUS, the TUS-I dataset only contains the homographs we methodically injected in order to study a specific effect on betweenness centrality. We ran our experiments by randomly selecting 50 pairs of values from different domains[8] and replaced them with our 50 injected homographs. Each experiment was repeated 4 times with a different seed for selecting the values for replacement. Since the number of homographs in our experiment is always 50, in an ideal scenario the top-50 BC scores would correspond to exactly those injected homographs.

We found that cardinality has the expected impact on BC scores in terms of separating homographs and unambiguous values. If the data values chosen for replacement have a not too small cardinality (i.e., they co-occur with many other values) then the BC score of their injected homograph was notably higher. We confirmed this observation in Table 2 where we varied the cardinality threshold for the data values chosen for replacement. Overall, as we increased the cardinality threshold, a larger percentage of the injected homographs ranked in the top-50. In fact, if the replaced values had a cardinality of 500 or higher, `DomainNet` consistently ranked at least 48 of the 50 injected homographs in the top 50. For reference, the largest attribute in TUS has 25,000 values and over half of all attributes have more than 500 values.

How does the number of meanings of a homograph affect homograph discovery? In addition to varying the cardinality of the replaced values, we also examined how the number of meanings of the injected homographs impacts their BC-based rankings. The number of meanings of an injected homograph is the number of values replaced for each injected homograph. The replaced values are all chosen from different domains to ensure that the injected homographs have consistently the specified amount of meanings. We explored injected homographs with the number of meanings in the range 2 to 8 for replaced data values with a cardinality of 500 or higher. Table 3 shows that as we increase the number of meanings, `DomainNet` becomes better at discovering them. This is consistent with our intuition for betweenness centrality since homographs with more meanings are more likely

[7] A common heuristic for the sample size is about 1-3% of the total number of nodes in the graph. This works well in practice with sparse graphs like `DomainNet` [17]. We will further test the validity of this heuristic in Section 5.4.

[8] Different domains in the TUS benchmark context means values from columns that are not unionable with each other.

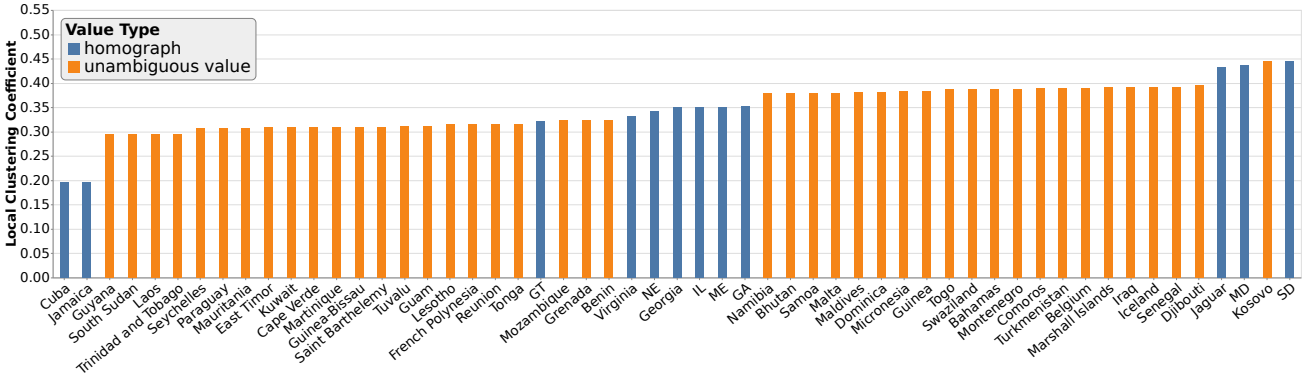


Figure 5: The top-55 data values with the lowest local clustering coefficients. Homographs are scattered throughout and do not necessarily have low LCC coefficients.

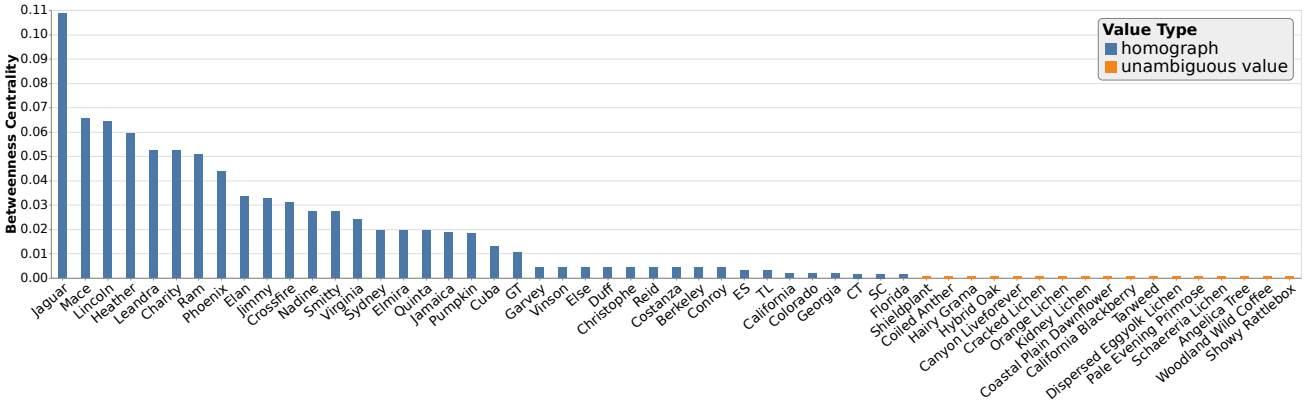


Figure 6: The top-55 data values with the greatest betweenness centrality scores. In the top-55 data values, 38 of them are homographs. The homographs not in the top-55 are country/state abbreviation homographs.

Table 3: % of injected homographs in the top 50 according to betweenness centrality while varying the number of meanings of the injected homographs

# meanings of injected homographs	2	3	4	5	6	7	8
% of homographs in top 50	97.5	97.5	98.5	98.5	100	100	100

to be hub nodes that connect multiple sets of nodes with each other in our bipartite graph representation of the data lake.

5.3 Homographs in TUS Benchmark

Lastly, we explore the performance of `DomainNet` with betweenness centrality on the real TUS dataset with its 26,035 real homographs. Since the number of homographs is large, we not only report precision, recall, and F1-score for the top-26,035 results, but for all top-k with k from 1 all the way to the number of nodes in our graph, i.e., 190,399. We do not compare against the D^4-based algorithm for homographs, because D^4 operates only on string attributes, and given the large number of numerical attributes the D^4 coverage will be even lower than in SB (where it only finds domains for 14 out of 39 attributes).

How does our approach perform on a real open-data benchmark? Figure 7 shows the summary of our top-k evaluation results. Notice that for relatively small values of k such as $k = 200$ our method can identify homograph values with high precision (0.89). Naturally, as we increase k precision decreases and recall

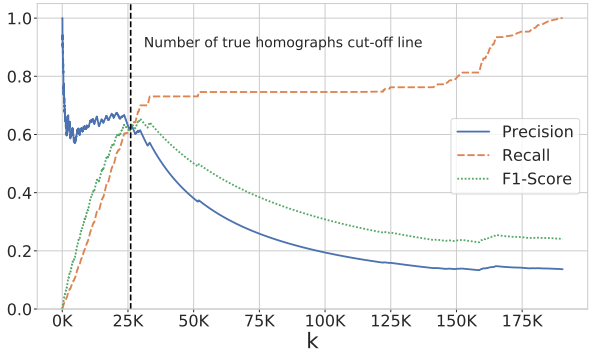


Figure 7: Top-k evaluation on the TUS dataset. The vertical line at k=26,035 denotes the number of true homographs in the dataset.

increases. At $k = 26,035$ (vertical line in Figure 7), which is the number of true homographs in the TUS benchmark, we achieve a precision, recall and F1-score of 0.622. The highest F1-score occurs at $k = 29,633$ where precision, recall, and F1-score are 0.615, 0.7 and 0.655, respectively.

It is important to emphasize that our approach is completely unsupervised and does not assume any external knowledge about the tables or their values. Existing state-of-the-art methods that tackle data integration tasks as described in Section 2 cannot be

readily used for homograph identification or their coverage is severely limited (e.g., knowledge-based approaches like AIDA [50]).

Below we report the top-10 values and their BC scores from the TUS benchmark.

- "Music Faculty" → 0.00064
- "Manitoba Hydro" → 0.00045
- "50" → 0.00029
- "1800ZZMALDY2" → 0.00028
- "." → 0.00027
- "Conseil de développement" → 0.00025
- "125" → 0.00023
- "2" → 0.00022
- "Biomedical Engineering" → 0.00022
- "SQA" → 0.00016

All 10 data values are homographs based on the ground truth. Notice that from a natural-language perspective these 10 values do not seem to be homographs, but a closer look at the data revealed good reasons why they were labeled as homographs. For example the value *Music Faculty* appears in two distinct contexts: as a geographic location/landmark in transportation-related tables as well as a department in university-related tables.

The value with the fifth-highest BC score is the period character. This may seem bizarre, but the period is used extensively as a null replacement in a large variety of tables and thus it acts as a homograph with a very large number of meanings. Finally, notice that we identify numerical values such as 50, 125 and 2, which appear in a variety of contexts such as addresses, identification numbers, quantity of products, etc. Numerical values are traditionally difficult to deal with in many data-integration tasks, hence being able to identify some of them in a completely unsupervised manner is a notable step toward better coverage for numerical values.

5.4 Scalability

As discussed in Section 3.4, Step 1 (graph construction) and Step 2 (centrality measure computation) are the most computationally expensive in our approach. In this section, we examine empirically the scalability of these steps.

The time to construct our bipartite graph is dependent on how long it takes to scan all input tables, which is a relatively fast operation. For example, the bipartite graph for the TUS dataset takes about 1.5 minutes to construct, which is how long it takes to read through each table in the dataset.

The runtime of Step 2 depends on the graph measure used. LCC is a local measure that is efficient to compute, but as we demonstrated in section 5.1 it is not as effective in finding homographs as BC is. Computing the LCC score for every node in the TUS dataset takes 4 seconds. For the global measure BC, since we are more interested in the score rankings rather than the scores themselves, approximating BC via sampling can significantly decrease the runtime without compromising quality.

In Figure 8, we examine how precision and runtime vary as we change the number of samples used for the approximate BC algorithm [17] on the TUS benchmark. Even for a small sample size (e.g., 1000), precision stabilises at 0.6. Notice that 1000 samples correspond to around .5% of the nodes in the TUS graph and it takes about 40 seconds for the algorithm to complete. The BC approximation has a complexity of $O(sm)$ where s is the number of nodes sampled and m the number of edges in the graph. Based on the literature and testing on our graphs we found that sampling 1% of the nodes provides a good approximation of BC that is very consistent with the score rankings produced by the exact BC computation.

We also considered a bigger data lake to further test execution times—the NYC education open data dataset as used in D^4 [36]. The bipartite graph representation of that dataset has roughly 1.5M nodes and 2.3M edges which is an order of magnitude larger than the bipartite graph for the TUS dataset. The graph was constructed in 3.5 minutes and the BC scores for every node were computed in 27 minutes using approximate BC on 1% of the nodes (~15K nodes).

To examine how runtime scales with graph size we extracted random subgraphs[9] of various sizes from the bipartite graph used for the NYC education dataset. We ran approximate BC for each graph by sampling 1% of its nodes and measured the runtime. Figure 9 shows that runtime increases linearly with graph size (i.e., number of edges) which is in accordance with the $O(sm)$ complexity of the approximate BC algorithm. .

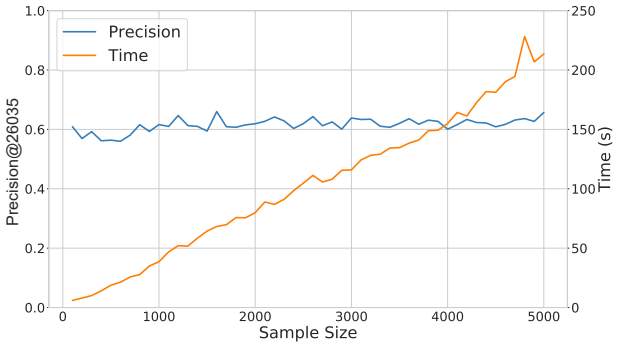


Figure 8: Precision at k (where k is the number of homographs in the dataset) and execution time at various sample sizes for approximate BC on the SB and TUS datasets. Exact BC on TUS took 150 minutes with a precision of 0.631.

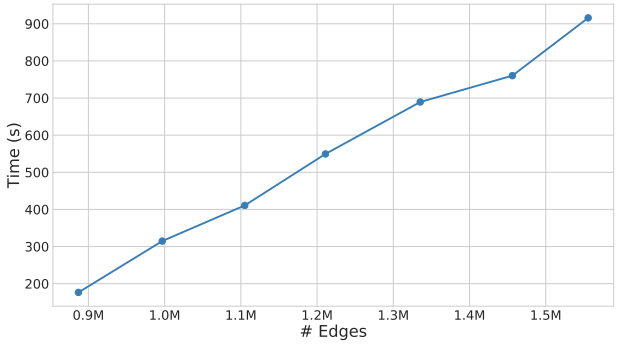


Figure 9: Runtime of approximate BC for various sized subgraphs based on the NYC education dataset.

5.5 Impact of homograph discovery on D^4

As shown in Table 1 the number of homographs in a real data lake can be large. To further understand the impact of homographs on

[9]The subgraphs were constructed by randomly selecting an attribute node and adding all its connecting value nodes. We repeat by selecting another attribute node until the subgraph reaches the desired size (within some margin)

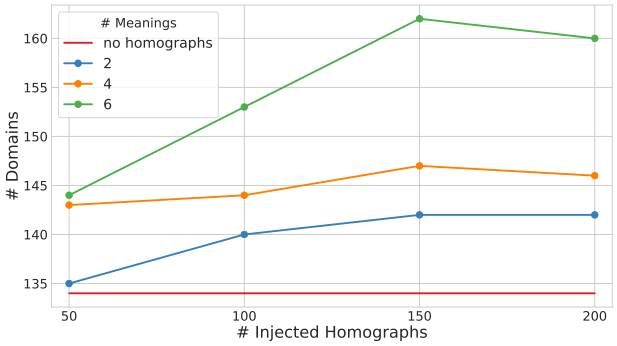


Figure 10: Number of domains found by D^4 over different TUS-injected datasets. The horizontal red line shows the number of domains found when no homographs were present in the dataset.

existing approaches, we consider the task of domain discovery and examine how knowing homographs *a priori* can benefit them.

We report the results of five different runs of D^4 in Figure 10. The plots show the number of domains found by D^4 (y-axis) as we vary the number and meanings of the injected homographs. To be fair in the comparison and to understand the impact of homographs on the domain discovery task, we use the TUS-I benchmark. We first ran D^4 over the dataset without homographs and then over the same dataset with injected homographs. More specifically, we injected 50, 100, 150 and 200 homographs with 2, 4 and 6 meanings. In all the above configurations the dataset always had 68 domains based on the ground truth. The horizontal line in Figure 10 shows that D^4 returns 134 domains for TUS-I with no homographs. The difference in the number of domains based on the ground truth and D^4's results is due to the nature of the TUS benchmark [33] as it is created from a set of large real open data tables that were randomly sliced vertically and horizontally. Consequently, in some cases the columns originating from the same table no longer share any values, causing D^4 to discover more domains than there are based on ground truth.

As we increase the number and meanings of the injected homographs, D^4 returns even more domains leading to lower accuracy. D^4's output provides statistics about the maximum and the average number of domains assigned to a column. In the TUS-I with no homographs, that maximum is 2 and the average is almost 1 (i.e., 1.031) and it increases with the number of homographs. With 200 homographs the maximum is 4 and the average is 1.04. We also ran D^4 on the TUS-I with 5000 injected homographs, to simulate a dataset with a large proportion of homographs as in the TUS benchmark. The maximum domains per column is 22 and the average is 1.7 with a total of 371 domains found. The presence of homographs is negatively affecting D^4 and causing it to erroneously assign larger numbers of heterogeneous domains to attributes as the number of homographs increases. Homograph discovery therefore is an important step that can be executed before domain discovery to improve its performance.

6 CONCLUSION AND FUTURE WORK

We presented `DomainNet`, a method for finding homographs in data lakes. To the best of our knowledge, this is the first solution for disambiguating data values in data lakes. Notably, our approach does not require complete or consistent attribute names.

We showed that a measure of centrality can effectively separate homographs from unambiguous values in a data lake by representing tables as a network of connections between values and attributes.

We compared against an alternative approach using D^4 to identify the semantic domain (type) of attributes [36] and labeling a value a homograph if it appears in more than one domain. Our direct computation of homographs has significantly better precision and recall than the domain-discovery approach. This seems to be due to D^4 at times placing homographs into a domain represented by their most popular meaning and the fact that D^4 does not find domains for every attribute. When we inject homographs into real data, `DomainNet` is robust to the number of meanings of the homographs, reliably finding homographs with even better accuracy as the number of meanings increases. We also demonstrated the importance of homograph detection by showing that the presence of homographs can have considerable impact on existing semantic integration tasks (specifically, domain discovery).

In a benchmark created from real data, our method provides a clear separation with high precision of homographs from values that are repeated, but always with the same meaning. The accuracy is influenced by the cardinality of the homograph (i.e., the number of data values with which the homograph co-occurs). When this number is too small, the bipartite graph representation is not always sufficient to effectively identify all homographs. In our experiments, the accuracy dropped from 97% to 85% as we reduced the cardinality of homographs.

The homographs we discover on real data include phrases with multiple meanings (e.g., `Music Faculty` referring both to a geographic location and to a University unit). They also include null values (e.g., a dot "." can indicate unknown/missing X where X varies in different contexts) and data errors (e.g., `Manitoba Hydro`, an electric company, is placed in the wrong column `Street Name`). In NLP, previous work on disambiguation primarily focuses on the disambiguation of words and named-entities. Our method is purely based on co-occurrence information and does not discriminate between different types of homographs. In fact, we provide the first approach to disambiguate numerical values in tables (e.g. 25 can be a street number or an ID number).

Identifying homographs from tables in a completely unsupervised manner can play an important role in improving other data-lake analysis tasks. Specifically, we are considering how to determine if a homograph is an error, e.g., the value has been placed in the wrong cell. With such knowledge, we can help not only identify such errors, but clean them as well. We also believe that our homograph metrics can improve supervised semantic type detection such as Sherlock [23] or SATO [51].

In this context, it will also be important to determine the number of distinct meanings of a homograph. Our approach is motivated by work on community detection where a community represents a meaning for a value (e.g., animal or car model). Hence we are investigating the role of community detection algorithms on discovery of meanings of values in data lake tables. Notice that in this problem, we do not know *a priori* what the communities are or even how many there are. Non-parameterized community detection algorithms can be used to discern the number of meanings of homographs. However, innovation is needed for homographs with large numbers of meanings (such as null equivalents) [21, and others].

To the best of our knowledge there are no available benchmarks for homograph detection. Our synthetic benchmark (SB)

and our benchmarks TUS and TUS-I (that use real open data tables [33]) are the first open benchmarks in this area.

In order to design a robust and completely unsupervised solution that scales to large data lakes, we have quite deliberately limited `DomainNet` to use only value co-occurrence information in table columns, ignoring additional structural information like co-occurrence of values in the same row. Our goal was to explore how much this information alone reveals about data value semantics. Given our strong positive results, we believe our metrics should become an important feature that could be used in other problems that involve understanding or integrating tables. An important open problem is to extend `DomainNet` to *collectively* resolve ambiguous metadata and data, perhaps using probabilistic graphical models that have been applied to collectively resolving multiple types of entities at once [26] and to collectively resolving data and metadata inconsistency in schema mapping [25].

Acknowledgments. This work was supported in part by the National Science Foundation (NSF) under award numbers IIS-1956096 and CAREER IIS-1762268.

REFERENCES

[1] R. Agerri and G. Rigau. Robust multilingual named entity recognition with shallow semi-supervised features. *Artif. Intell.*, 238:63–82, 2016.
[2] S. Arora and S. Bedathur. On embeddings in relational databases. *CoRR*, abs/2005.06437, 2020.
[3] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Trans. Knowl. Discov. Data*, 1(1):5, 2007.
[4] I. Bhattacharya, L. Getoor, and Y. Bengio. Unsupervised sense disambiguation using bilingual probabilistic models. In *ACL*, pages 287–294. ACL, 2004.
[5] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, pages 1247–1250. ACM, 2008.
[6] U. Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.
[7] R. Cappuzzo, P. Papotti, and S. Thirumuruganathan. Creating embeddings of heterogeneous relational datasets for data integration tasks. In *SIGMOD*, pages 1335–1349, 2020.
[8] T. Chakraborty, A. Dalmia, A. Mukherjee, and N. Ganguly. Metrics for community analysis: A survey. *ACM Comput. Surv.*, 50(4):54:1–54:37, 2017.
[9] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, and K. Stefanidis. An overview of end-to-end entity resolution for big data. *ACM Comput. Surv.*, 53(6), 2020.
[10] Y. Dong, K. Takeoka, C. Xiao, and M. Oyamada. Efficient joinable table discovery in data lakes: A high-dimensional similarity-based approach. *CoRR*, abs/2010.13273, 2020.
[11] J. Eberius, P. Damme, K. Braunschweig, M. Thiele, and W. Lehner. Publish-time data integration for open data platforms. In *Proceedings of the 2nd International Workshop on Open Data (WOD)*, pages 1:1–1:6, 2013.
[12] J. Eberius, M. Thiele, K. Braunschweig, and W. Lehner. Top-k entity augmentation using consistent set covering. In *SSDBM*, pages 8:1–8:12. ACM, 2015.
[13] R. C. Fernandez, J. Min, D. Nava, and S. Madden. Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In *ICDE*, pages 1190–1201. IEEE, 2019.
[14] A. A. Ferreira, M. A. Gonçalves, and A. H. F. Laender. A brief survey of automatic methods for author name disambiguation. *SIGMOD Rec.*, 41(2):15–26, 2012.
[15] S. Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.
[16] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.
[17] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, pages 90–100. SIAM, 2008.
[18] L. Getoor and A. Machanavajjhala. Entity resolution: Theory, practice & open challenges. *PVLDB*, 5(12):2018–2019, 2012.
[19] A. Goel, C. A. Knoblock, and K. Lerman. Exploiting structure within data for accurate labeling using conditional random fields. In *Proceedings of the 14th International Conference on Artificial Intelligence (ICAI)*, 2012.
[20] O. Hassanzadeh, M. J. Ward, M. Rodriguez-Muro, and K. Srinivas. Understanding a large corpus of web tables through matching with knowledge bases: an empirical study. In *Proceedings of the 10th International Workshop on Ontology Matching*, volume 1545 of *CEUR Workshop Proceedings*, pages 25–34. CEUR-WS.org, 2015.
[21] K. Henderson, T. Eliassi-Rad, S. Papadimitriou, and C. Faloutsos. HCDF: A hybrid community discovery framework. In *SIAM International Conference on Data Mining, SDM*, pages 754–765. SIAM, 2010.
[22] K. Z. Hu, S. N. S. Gaikwad, M. Hulsebos, M. A. Bakker, E. Zgraggen, C. A. Hidalgo, T. Kraska, G. Li, A. Satyanarayan, and Ç. Demiralp. Viznet: Towards A large-scale visualization learning and benchmarking repository. In *CHI*, page 662. ACM, 2019.
[23] M. Hulsebos, K. Z. Hu, M. A. Bakker, E. Zgraggen, A. Satyanarayan, T. Kraska, Ç. Demiralp, and C. A. Hidalgo. Sherlock: A deep learning approach to semantic data type detection. In *SIGKDD*, pages 1500–1508. ACM, 2019.
[24] I. Iacobacci, M. T. Pilehvar, and R. Navigli. Embeddings for word sense disambiguation: An evaluation study. In *ACL (1)*. ACL, 2016.
[25] A. Kimmig, A. Memory, R. J. Miller, and L. Getoor. A collective, probabilistic approach to schema mapping using diverse noisy evidence. *IEEE Trans. Knowl. Data Eng.*, 31(8):1426–1439, 2019.
[26] P. Kouki, J. Pujara, C. Marcum, L. M. Koehly, and L. Getoor. Collective entity resolution in multi-relational familial networks. *Knowl. Inf. Syst.*, 61(3):1547–1581, 2019.
[27] C. Koutras, M. Fragkoulis, A. Katsifodimos, and C. Lofi. REMA: graph embeddings-based relational schema matching. In *EDBT*, volume 2578, 2020.
[28] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.
[29] O. Lehmberg, D. Ritze, R. Meusel, and C. Bizer. A large public corpus of web tables containing time and context metadata. In *WWW*, pages 75–76, 2016.
[30] F. H. Levin and C. A. Heuser. Evaluating the use of social networks in author name disambiguation in digital libraries. *J. Inf. Data Manag.*, 1(2):183–198, 2010.
[31] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. *PVLDB*, 3(1):1338–1347, 2010.
[32] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena. Data lake management: Challenges and opportunities. *PVLDB*, 12(12):1986–1989, 2019.
[33] F. Nargesian, E. Zhu, K. Q. Pu, and R. J. Miller. Table union search on open data. *PVLDB*, 11(7):813–825, 2018.
[34] R. Navigli. Word sense disambiguation: A survey. *ACM Comput. Surv.*, 41(2):10:1–10:69, 2009.
[35] M. E. J. Newman. *Networks: An Introduction*. Oxford University Press, 2010.
[36] M. Ota, H. Mueller, J. Freire, and D. Srivastava. Data-driven domain discovery for structured datasets. *PVLDB*, 13(7):953–965, 2020.
[37] G. Papadakis, G. M. Mandilaras, L. Gagliardelli, G. Simonini, E. Thanos, G. Giannakopoulos, S. Bergamaschi, T. Palpanas, and M. Koubarakis. Three-dimensional entity resolution with JedAI. *Inf. Syst.*, 93:101565, 2020.
[38] G. Papadakis, J. Svirsky, A. Gal, and T. Palpanas. Comparative analysis of approximate blocking techniques for entity resolution. *PVLDB*, 9(9):684–695, 2016.
[39] M. T. Pilehvar and R. Navigli. A large-scale pseudoword-based evaluation framework for state-of-the-art word sense disambiguation. *Comput. Linguistics*, 40(4):837–881, 2014.
[40] M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. *Data Min. Knowl. Discov.*, 30(2):438–475, 2016.
[41] D. Ritze, O. Lehmberg, Y. Oulabi, and C. Bizer. Profiling the potential of web tables for augmenting cross-domain knowledge bases. In *WWW*, pages 251–261. ACM, 2016.
[42] G. Simonini, S. Bergamaschi, and H. V. Jagadish. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *PVLDB*, 9(12):1173–1184, 2016.
[43] B. Skaggs and L. Getoor. Topic modeling for wikipedia link disambiguation. *ACM Trans. Inf. Syst.*, 32(3):10:1–10:24, 2014.
[44] N. R. Smalheiser and V. I. Torvik. Author name disambiguation. *Annu. Rev. Inf. Sci. Technol.*, 43(1):1–43, 2009.
[45] C. L. Staudt, A. Sazonovs, and H. Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Netw. Sci.*, 4(4):508–530, 2016.
[46] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.
[47] K. Takeoka, M. Oyamada, S. Nakadai, and T. Okadome. Meimei: An efficient probabilistic approach for semantically annotating tables. In *AAAI*, pages 281–288. AAAI Press, 2019.
[48] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.
[49] V. Yadav and S. Bethard. A survey on recent advances in named entity recognition from deep learning models. In *COLING*, pages 2145–2158. ACL, 2018.
[50] M. A. Yosef, J. Hoffart, I. Bordino, M. Spaniol, and G. Weikum. Aida: An online tool for accurate disambiguation of named entities in text and tables. *PVLDB*, 4(12):1450–1453, 2011.
[51] D. Zhang, Y. Suhara, J. Li, M. Hulsebos, Ç. Demiralp, and W. Tan. Sato: Contextual semantic type detection in tables. *PVLDB*, 13(11):1835–1848, 2020.
[52] E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller. LSH ensemble: Internet-scale domain search. *PVLDB*, 9(12):1185–1196, 2016.

GPU-INSCY: A GPU-Parallel Algorithm and Tree Structure for Efficient Density-based Subspace Clustering

Jakob Rødsgaard Jørgensen
Department of Computer Science
Aarhus University, Denmark
jakobrj@cs.au.dk

Katrine Scheel
Department of Computer Science
Aarhus University, Denmark
scheel@cs.au.dk

Ira Assent
Department of Computer Science
DIGIT Aarhus University Centre for Digitalisation, Big Data and Data Analytics
Aarhus University, Denmark
ira@cs.au.dk

ABSTRACT

Subspace clustering is the task of grouping objects based on mutual similarity in subspaces of the full-dimensional space. The INSCY algorithm extends the well-known density-based clustering algorithm DBSCAN. It finds dimensionality-unbiased non-redundant subspace clusters using a tree structure to speed up the processing of subspaces. Still, finding density-based clusters in all subspaces implies an exponential search space in the number of dimensions. Thus, the running time of INSCY is still measured in hours on even small datasets of 2000 points. For larger datasets, it becomes prohibitively expensive.

To benefit from INSCY for real-world sized datasets, we propose a novel GPU-parallel approach that runs on standard graphics cards. To utilize the many cores of the GPU, we need new algorithmic strategies that fit the computational model of the GPU. While the GPU provides a large number of threads, traditional algorithms incur diverging threads and poor memory alignment, both of which lead to idle time and poor runtime performance. In INSCY, extracting subspace regions from the SCY-tree structure and the density-based clustering of regions itself are thus unfit for the GPU.

Our novel GPU-friendly algorithm GPU-INSCY computes the same subspace clustering as INSCY at dramatically reduced runtimes. To achieve this, we devise a restructured SCY-tree index-structure and associated operations for the GPU, as well as a GPU-parallel density-based subspace clustering.

We experimentally show that GPU-INSCY scales well with the size of the dataset and the number of dimensions, and improves the running time of INSCY by a factor of several thousand for large datasets of high dimensionality.

1 INTRODUCTION

Clustering, i.e., grouping data points based on mutual similarity, is a widely used data mining task, e.g., for grouping customers to allow for targeted marketing. However, real-world data is often high-dimensional, and a higher number of dimensions means that there are more possibilities for points to seem dissimilar. This is known as the curse of dimensionality. Due to this effect, points tend to group within a subspace of the full-dimensional space, leading to the task of subspace clustering [2, 4, 15], where we search for clusters with all possible subspaces. To search for such clusters, we often employ density-based clustering similar to DBSCAN [12]. Most subspace clustering algorithms, e.g., SUBCLU [15], use a fixed density threshold independent of the subspace's dimensionality. When finding clusters, the density threshold needs to match the expected density such that we can find all points within clusters, but without including everything. However, the expected density is lower for higher-dimensional subspaces than it is for lower-dimensional subspaces. For density-based subspace clustering, this problem implies that density-measures that do not take the subspace's dimensionality into account are biased toward lower subspaces. To address this problem, Assent et al. [6] formulated a dimensionality-unbiased density-measure and utilized this in the algorithm INSCY [8]. INSCY, furthermore, removes redundancy and provides an index-structure called SCY-tree used to partition and prune regions of density-connected data points. A drawback that remains, is that the running time is still measured in hours on even small datasets of a couple of thousands of points.

To reduce the runtime of dimensionality-unbiased density-based subspace clustering, we exploit modern graphics cards (GPUs), capable of general-purpose computations, fast context switches, and parallelizing over many cores, but with a restrictive computational model and limited memory. The high computational throughput of GPUs has been utilized to improve clustering runtimes [1, 5, 10]. However, to our knowledge, there exists no GPU-parallelization of a dimensionality-unbiased index-supported algorithm like INSCY, which is challenging to GPU-parallelize due to index and depth-first subspace search being optimized for (sequential) CPU processing.

Contributions. In this work, we present a novel GPU-parallel algorithm, called GPU-INSCY, which provides the same clusterings as INSCY at substantially reduced runtimes. To achieve this, we restructure several major parts of INSCY, the index-structure SCY-tree, the operations used to partition regions of data, and the clustering of points. INSCY partitions regions represented by SCY-trees through a sequence of operations. We show how to make these operations parallel and combine several partitions into one process. Combining these allows us to avoid many redundant iterations and temporary copies. The clustering step is also GPU-parallelized and improved further by utilizing the density monotonicity for neighborhoods in increasing subspaces.

This paper is organized as follows: Section 2 discusses related work, Section 3 gives the background of subspace clustering and INSCY, Section 4 describes our new parallel algorithm GPU-INSCY, Section 5 presents the experimental comparison of INSCY and GPU-INSCY, and Section 6 concludes our work.

2 RELATED WORK

Subspace clustering is the task of grouping points based on mutual similarity in any possible subspace of the full-dimensional space, hence its worst case complexity is exponential in the number of dimensions.

Algorithms for subspace clustering [2, 4, 6–8, 11, 14, 15, 25] are often categorized into bottom-up or top-down approaches [16, 21, 24, 26]. Bottom-up approaches start with clustering in 1-dimensional subspaces, iteratively combining k-dimensional subspace clusters into $(k+1)$-dimensional subspace clusters. CLIQUE [4] and MAFIA [14] are grid-based approaches that may miss subspace clusters spanning across grid cells. Instead of clustering dense cells, SUBCLU [15] clusters dense points, as in the density-based full space clustering algorithm DBSCAN[12]. An issue with SUBCLU and other density-based subspace clustering approaches is that they use a fixed density-threshold for all subspaces. Therefore, they do not take dimensionality into account and are biased towards lower-dimensional subspace clusters. INSCY is an extension of SUBCLU that mitigates this problem by introducing a density measure normalized by a subspace's expected density.

Top-down approaches start by clustering the full-dimensional space and iteratively refine the subset of dimensions associated with each subspace cluster [2, 3, 29]. These approaches limit subspace clusters by assigning each point in the data to exactly one subspace cluster. Due to the exponential search for subspaces, many of the algorithms take an approximate approach to subspace clustering [2, 14, 20]. They do so using a heuristic to pick the subspaces that are examined or only compute clusterings of dense regions instead of single dense points. These approaches might miss clusters that exact algorithms like INSCY capture.

Even though exact subspace clustering algorithms are time consuming, few algorithms have been proposed to reduce the running time by exploiting the high computational throughput of the GPU. Utilizing the many cores of the GPU is highly challenging because of the distinct and limited computational model, as well as limited memory. There have been proposed several GPU-parallelized full-space clustering algorithms [5, 10, 13, 17, 19]. One of the earliest GPU versions of the full-space clustering algorithm DBSCAN was CUDA-DClust* [10], which starts multiple searches for clusters in parallel. If multiple searches start within the same cluster, they are merged. Multiple other GPU-versions of DBSCAN have been developed [5, 18, 19, 28]. Our assessment of self-reported results suggest that G-DBSCAN[5] and CUDA-DClust*[10] are the best performing options. An experimental evaluation [22] studies three of these GPU-versions and finds that G-DBSCAN is the fastest and CUDA-DClust* uses less memory.

Only one GPU-parallelization of a well-known subspace clustering approach has been proposed [1] for grid-based MAFIA. GPUMAFIA parallelizes one operation at a time, mapping nested for-loops of minor computations directly to parallel threads. Our restructuring of INSCY lets us GPU-parallelize GPU-INSCY even further such that we can even parallelize operations performed at different points of the process. We completely restructure the algorithm and its underlying SCY-tree structure to fit the computational model and the memory structure of the GPU.

To the best of our knowledge, we are the first to develop a GPU-parallelized version of a density-based subspace clustering algorithm, in particular an algorithm that supports dimensionality-unbiased density measures and exploits indexing structures for efficient computation.

3 BACKGROUND

3.1 The graphics processing unit

We give a short introduction to graphics processing units (GPUs) and their computational model. When using a GPU for general-purpose computation, the GPU is *co-processor*, and the CPU is *main processor*. Throughout the paper, we use the term *parallel* to denote parallel execution under the GPU's computational model. The main difference between a multi-core CPU and a GPU is that GPUs can perform fast context switches and that several cores on the GPU uses the same program counter and, therefore, must perform the same operations.

CUDA is NVIDIA's framework for using their line of GPUs. It uses the concept of a kernel, which is a function executed on multiple threads in parallel. Threads are organized into blocks, and all threads within a single block are capable of synchronizing, share fast accessible memory, and use atomic operations. However, there is a physical limit to the number of threads a block can contain, and the communication between threads comes at a time-cost. Each block is further separated into warps. All threads within a warp share a program counter, implying that they must perform the same instructions (SIMD) at all times. In the case of branch-diversion, threads in different branches will remain idle until the other branch has finished.

When parallelizing operations on the GPU, we are not guaranteed any order of executions. Therefore, our goal is to identify *independent* operations, i.e., operations that do not use the partial result of each other and therefore can be run in any order without changing the final result. All allocation of memory and calls to kernels are done by the CPU and executed on the GPU. All communication with the GPU comes with a time-cost due to the large latency of data transfer. Therefore, it is essential to balance where data is processed and how long it takes to transfer.

3.2 INSCY

We describe INSCY briefly. For further details please see [8]. We use the following terminology: let $X \in \mathbb{R}^{n \times d}$ be a d-dimensional dataset with n points, $D = \{0, \ldots, d-1\}$ an index set for the full dimensional space, $S \subseteq D$ a subspace of D, and $N_\varepsilon^S(p)$ the neighborhood with radius ε of a point p in subspace S.

According to INSCY [6], a subspace cluster is a maximal set of points of at least min_C, which are density-connected in a subspace according to some density measure, and which is not redundant w.r.t. a higher dimensional subspace projection:

Definition 3.1. **INSCY Subspace Cluster**
A set of points $C \subseteq X$ in subspace $S \subseteq D$ is a subspace cluster if:

- objects in C are **S-connected**: $\forall p, q \in C : \exists o_1, \ldots, o_m \in C : p = o_1 \wedge q = o_m \wedge \forall i \in \{2, \ldots, m\} : o_i \in N_\varepsilon^S(o_{i-1})$
- all points fulfill the **density criterion**: $\forall p \in C : dc^S(p)$,
- C is **maximal**, i.e., contains all S-connected objects: $\forall p, q \in X : p, q$ S-connected $\Rightarrow p \in C \wedge q \in C$,
- **minimum cluster size**: $|C| \geq min_C$,
- **not redundant**: $\nexists C', S'$ subspace cluster with $C' \subseteq C \wedge S \subset S' \wedge |C'| \geq r \times |C|$

where r is the redundancy parameter, min_C is the minimum size of a cluster, and $dc^S(p)$ is any dimensionality-unbiased density criterion within subspace S.

In this paper, we use the dimensionality-unbiased rectangular density measure for the density criterion $dc^S(p) := |N_\varepsilon^S(p)| \geq \max(F \cdot \alpha(S), \mu)$, where F is the density factor threshold, $\alpha(S) = \mathbb{E}_S\left[|N_\varepsilon^S(p)|\right] = |X| \frac{c(S) \times \varepsilon^{|S|}}{v_S}$ is the expected density, $c(S) = \pi^{\frac{|S|}{2}} / \Gamma\left(\frac{|S|}{2} + 1\right)$ with $\Gamma(n+1) = n \times \Gamma(n), \Gamma(1) = 1, \Gamma(1/2) = \sqrt{\pi}$, v_S is the volume of subspace S, and μ is the minimum number of points required for not just being pseudodense. Other density measures can also be used. For further details see [6]. Note that

Def. 3.1 is similar to density-based clustering in DBSCAN [12], but with an unbiased density notion wrt. subspaces.

SUBCLU [15] uses monotonicity of density-connectivity to prune points that lie outside clusters in a lower-dimensional subspace projection. However, for INSCY's unbiased density measure that scales with the expected density of a subspace, monotonicity is lost. Still, as [6] observes, pruning can be done by discarding points that are not dense w.r.t. the lowest possible density threshold, i.e., for the full-space. INSCY finds such points, called not weak-dense, which can safely be pruned before searching for clusters within superspaces of the current space. A point is weak-dense if $|N_\varepsilon^S(p)| \geq \max(F \times \alpha(D), \mu)$.

3.2.1 The INSCY algorithm. The idea of INSCY is to bound the search for subspace clusters by identifying regions that fully contain potential clusters. INSCY describes such a region by the dimensions it spans and the respective intervals in these dimensions, and call it a subspace region. INSCY performs a depth-first search (DFS) of the subspace regions, i.e., enumerating all possible subspace regions. INSCY does so by recursively extending with one dimension at a time and partitioning the region into intervals along that dimension. When INSCY returns from the recursion, it performs density-based clustering within the current subspace region to obtain the clusters. This implies that INSCY cluster points within all superspaces of the current space first.

Each dimension is partitioned into a fixed number of cells. As a cluster likely spans multiple cells, INSCY register this by having a border between each cell at the size of the neighborhood radius ε. When performing density-based clustering, it follows that if there are no points within this border, the two cells' points cannot be density-connected. Otherwise, a cluster may span both cells. Such connected cells are referred to as S-connected. S-connected cells must be merged into a density-connected interval to ensure that no clusters are split. An interval spanning multiple cells is identified by the first cell. A dimension might have multiple density-connected intervals, and INSCY is called recursively on each interval in a depth-first manner. The whole process of expanding with a new dimension and bounding to a density-connected interval is referred to as restricting w.r.t. a new dimension and the cell identifying the interval. The pair of dimension d and cell c is called a descriptor (d, c). When expanding with a new dimension, we expand one region at a time. Figure 1 shows a 1-dimensional example, and the expansion into two dimensions. On the left, the dimension is split into three cells, where two are S-connected and merged into one interval marked by green. On the right, we see the expansion. The red region is split into cells along the added dimension and connected with any S-connected cells, and likewise for the green region.

To keep track of the possible dimensions and cells that can be restricted, INSCY introduces an index-structure called SCY-tree. The idea of SCY-tree is to precompute the number of points within cells along a dimension such that restricting becomes easier. The

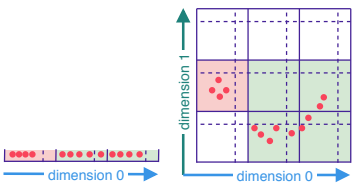


Figure 1: Expansion of 1-d regions into 2-d

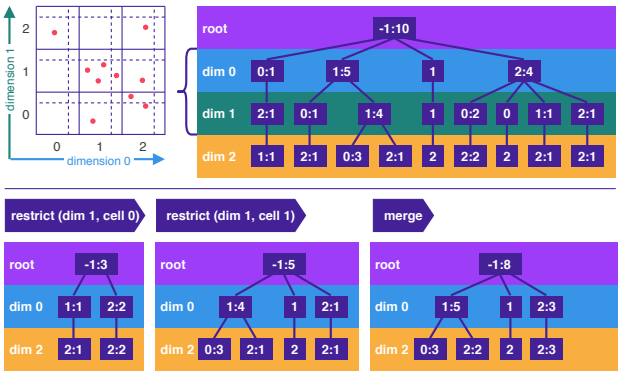


Figure 2: SCY-tree for examples in [8]; node values *cell* : *count*; dimensions and points colored as in later figures

SCY-tree, therefore, represents the dimensions and cells not yet restricted. The SCY-tree is a tree-structure containing nodes that represent a partition of a space along a specific dimension. All nodes regarding a specific dimension are located at the same height in the SCY-tree, which we call a layer. The children of a node represent splits into cells along a dimension, one child per cell. Each node contains its cell number and the count of points within the cell it represents. A cell with an S-connection is represented by adding a sibling with the same cell number, but with the count of points set to -1. Such a node is called an S-connector node. INSCY keeps track of S-connections by continuing the path of S-connector nodes down to the leaf layer. The root node of the SCY-tree represents a restricted subspace region. SCY-trees that represent regions that share a border are called neighboring SCY-trees. For further details, see [8].

Figure 2 (top) shows an example of an initial SCY-tree for the full-dimensional space. In this example, the space is first partitioned along dimension 0, creating three cells noted by the cell number and the count of points in that cell *cell* : *count*. Cell 1 has an S-connection, which is represented by a node without a count of points. Each cell is then further partitioned along dimension 1, discarding cells that do not contain any points.

INSCY proceeds as in Algorithm 1. For each descriptor, create a restricted SCY-tree. If cells in the SCY-tree are S-connected, merge connected restricted SCY-trees into one final restricted SCY-tree. INSCY prune the final restricted SCY-tree for redundancy, call recursively, and cluster the points if there is a possibility for non-redundant clusters.

Restrict. INSCY restricts a SCY-tree by identifying nodes matching the current descriptor, i.e., the nodes residing on the layer of the restricted dimension and with the same cell number as the descriptor. For each matching node, copy the node's path to the root and subtrees below the node into a new restricted

Algorithm 1 INSCY($scytree$, d_f, X, d, r, F, μ, ε, min_C, R)

```
1: for d_re = d_f to d do
2:     for c_re = 0 to n_cells do
3:         scytree' ← restrict(scytree, d_re, c_re)
4:         scytree' ← mergeNeighbors(scytree, scytree', d_re, c_re)
5:         if prune_recursion(scytree', F, μ, ε, min_C) then
6:             INSCY(scytree', d_re + 1, X, d, r, F, μ, ε, min_C, R)
7:             if prune_redundancy(scytree', r, R) then
8:                 R ← R ∪ clustering(scytree', X, F, μ, ε)
```

SCY-tree. Since the SCY-tree keeps track of not yet restricted dimensions, the matching node itself is not copied. The node's children are now children of the node's parent. The count of points is also updated to reflect the number of points in the restricted region. Figure 2 (bottom) contains two restricted SCY-trees for descriptors $(1, 0)$ and $(1, 1)$ and the merged result. For descriptor $(1, 0)$ only 2 nodes match, leading to a small SCY-tree.

Merge. INSCY merges neighboring restricted SCY-trees if there exists an S-connection, i.e. when an S-connector path starts at dimension d and has cell number c that matches the current descriptor (d, c). Merge is done by going through the two restricted SCY-trees and copying the nodes in both. A node can be represented in several SCY-trees. During the merge, nodes with the same cell number and the same parent are merged. Figure 2 (bottom), shows that the descriptor $(1, 0)$ matches an S-connector node, the node represented by only a 0 on dimension 1, and therefore INSCY restricts the neighboring descriptor $(1, 1)$ and merges the two restricted SCY-trees.

Pruning recursion. To reduce the search space, INSCY prunes the final restricted SCY-tree before calling recursively, as follows: Remove non-weak dense points and check if the region's number of points still exceeds min_C. INSCY only proceeds with the recursion if this is the case, as further restrictions will only reduce the number of points.

Pruning redundancy. When returning from the recursive call INSCY has found clusters in all superspaces of the current subspace. The current region can therefore be pruned by redundancy. INSCY prunes by redundancy by checking if the result already contains a cluster covering a factor r of the points in the restricted region. If the number of points in the region is large enough, INSCY computes the density-based clustering on all points in the final restricted SCY-tree and adds all non-redundant clusters to the result.

4 GPU-INSCY ALGORITHM

INSCY is inherently computationally expensive, making it infeasible to run on large real-world datasets. As mentioned in the introduction, GPUs provide computational power that algorithms designed for a different computational model of single-core CPUs, as INSCY, cannot utilize. We design an algorithm for the GPU that reduces the running time of INSCY substantially, making it feasible to run on much larger datasets. To summarize the notation found in this section we provide Table 1 for ease of reading. Recall that threads in a warp must execute the same instructions to fully utilize the GPU's computational power. INSCY does not group similar operations and would perform poorly on the GPU.

The idea of each iteration in INSCY is to bound a subspace region by restricting and merging, prune that region, and perform clustering in that region. This process is repeated until all clusters in all subspace regions are found. This approach is efficient for a sequential algorithm. However, when parallelizing for the GPU, we prefer grouping identical and independent operations to make each kernel call utilize as many cores as possible. Making INSCY run parallel on the GPU is not straightforward since many partial computations depend on previous results. E.g., in the alternation between restricting and merging SCY-trees, we need the previous merged SCY-tree and the neighboring restricted SCY-tree before continuing to merge.

Table 1: Notation

n_{nodes}	number of nodes
n_{pts}	number of points
n_{cells}	number of cells
n_{dims}	number of dimensions
n_{r_dims}	number of restricted dims
$pa \in \mathbb{N}^{n_{nodes}}$	parent array
$ce \in \mathbb{N}^{n_{nodes}}$	cell array
$co \in \mathbb{N}^{n_{nodes}}$	count array
$la \in \mathbb{N}^{n_{dims}}$	layer-indexing array
$dims \in \mathbb{N}^{n_{dims}}$	dimension array
$r\_dims \in \mathbb{N}^{n_{r\_dims}}$	restricted dims array
$po \in \mathbb{N}^{n_{pts}}$	point-id array
$pl \in \mathbb{N}^{n_{pts}}$	point-placement array
$incl \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells} \times n_{nodes}}$	node inclusion array
$incl_{pts} \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells} \times n_{pts}}$	point inclusion array
$idx \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$	node new-index array
$idx_{pts} \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{pts}}$	point new-index array
$n\_co \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$	new-count array
$is\_S(i)$	is S-connection
$s\_incl(j, i, c)$	should be included
$S \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells}}$	S-connection array
$M \in \mathbb{N}^{n_{dims} \times n_{cells}}$	merge map
$n\_pa \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$	new-parent array
$n\_ch \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes} \times n_{cells} \times 2}$	new-children array
$rep(j, c, i)$	representative node

Algorithm 2 GPU-INSCY($scytree'$, d_f, X, d, r, F, μ, ε, min_C, R)

```
1:  L ← GPU_restrict_and_merge(scytree, d_f, d)
2:  precompute_neighborhoods(X, L, ε)
3:  for d_re ← d_f to d − d_f do
4:      C ← 1d array of size |X| initialized to −1
5:      for ∀scytree' ∈ L[d_re] do
6:          if prune_recursion(scytree', F, μ, ε, min_C) then
7:              GPU-INSCY(scytree', d_re + 1, X, d, r, F, μ, ε, min_C, R)
8:              if prune_redundancy(scytree', r, R) then
9:                  L' ← L' ∪ {(scytree', C)}
10: R ← R ∪ GPU_clustering(L', X, F, μ, ε)
```

In this section, we present a new algorithm called GPU-INSCY, in which we tackle the problem of identifying and reorganizing the operations that can be performed in parallel to reduce running time. Contrary to INSCY, GPU-INSCY aims to perform similar and independent operations simultaneously for multiple final restricted SCY-trees to utilize multiple thread blocks. Remember that this allows us to use more cores, but it is only possible if the threads in different blocks do not need to communicate.

We first outline the general order of computations in GPU-INSCY, and we later explain this reordering. These reorderings do not affect the result since the reordered operations are independent of each other as discussed below for each change we introduce. GPU-INSCY can be seen in Algorithm 2. First, compute the set L of all final restricted SCY-trees. Precompute the neighborhoods for all points in all final restricted SCY-trees. For each final restricted SCY-tree, prune the recursion, call GPU-INSCY recursively, and prune for redundancy. All non-pruned final restricted SCY-trees are added to L'. Finally, we cluster all points in each of the final restricted SCY-trees in L'.

Restrict and merge. In GPU-INSCY, we isolate all restrict and merge operations at the beginning of the algorithm, whereas INSCY performs them ad hoc. We isolate the operations such that we can parallelize them in different thread blocks. The result of

each restrict and merge operation only depends on the information parsed to the recursion. Computing all restricted SCY-trees at the beginning does, therefore, not change the final result. Parallelizing within each thread block is not a simple task due to both the alternation between restrict and merge and the fact that INSCY only visits nodes in the SCY-trees one by one when restricting and merging. We discuss how to parallelize restrict and merge in Section 4.1.2, after introducing a representation of the SCY-tree index-structure for the GPU in Section 4.1.1.

Precomputing the neighborhoods. Computing the neighborhoods is an expensive task, and it is used both for the clustering and when computing weak-density while pruning a recursion. In Section 4.2, we describe how to precompute the neighborhoods in parallel and how we take advantage of having direct access to the neighborhoods in a subspace of the current space.

Pruning. In Section 4.3, we parallelize both pruning phases following the same approach as for restrict and merge.

Clustering. In Section 4.2, we change the sequential way of expanding the clusters [12] with one density-connected point at a time, to obtain a more efficient clustering algorithm.

4.1 SCY-tree on the GPU

The SCY-tree representation and the associated operations are not very suited for the GPU. Section 4.1.1 describes how to represent the SCY-tree in a GPU friendly fashion and Section 4.1.2 describes how to perform the restrict and merge operations in parallel.

4.1.1 Representing the SCY-tree on the GPU. Handling memory on the GPU is more restrictive than on the CPU, and allocating memory can only be done from the CPU. Furthermore, it is expensive to alternate between calling kernels, transferring data, and allocating memory. Therefore, we prefer to allocate memory and transfer data as few times as possible. GPU memory is loaded one block at a time to reduce latency, implying that data used close together in time should be placed close together in memory. If the data we use is not placed in the same block, we get cache misses, i.e., not using the loaded data, which we would like to reduce. For ease of reference, we call the GPU friendly representation of the SCY-tree GPU-SCY-tree. A way to represent tree structures on the CPU is to create an object for each node with pointers to its children, parent, and other values in the tree. This structure is very flexible and allows adding nodes on the fly. However, this does not fit well with the restrictions on the GPU.

Remember, all nodes for a particular dimension are placed on the same layer in the SCY-tree. These layers are indexed by j starting with $j = -1$ for the root and incrementing toward the leaf layer $j = n_{dims} - 1$, implying that lower indices are above the higher indices in the SCY-tree. In Section 4.1.2, we describe how we handle all nodes on the same layer simultaneously, and we would therefore like to place these nodes close together in memory. The same is the case for points contained in the tree.

Instead of representing nodes as objects, we choose to represent the GPU-SCY-tree as arrays, with an entry for each node. Each array represents the kind of pointer or values that a node contains. In the arrays, we locate nodes on the same layer in the SCY-tree next to each other and order the layers by their index j. In this way, data for nodes on the same layer is placed close together in memory, making it more likely to avoid cache-misses. We organize points using the same reasoning. To represent the GPU-SCY-tree, we use a total of eight arrays with one entry per node, point, or dimension. An example is given in Figure 3. Besides the arrays we also keep count of the number of nodes

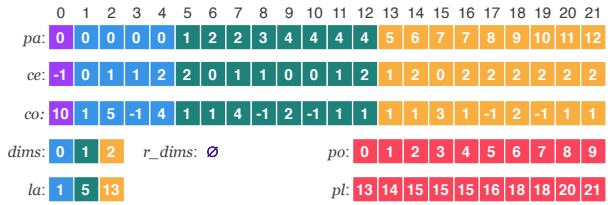


Figure 3: GPU-SCY-tree for SCY-tree in Figure 2.

n_{nodes}, number of points n_{pts}, number of cells n_{cells}, number of dimensions in the SCY-tree n_{dims}, and number of restricted dimensions n_{r_dims}.

The nodes are represented using three arrays: the parent pointer $pa \in \mathbb{N}^{n_{nodes}}$, the cell number $ce \in \mathbb{N}^{n_{nodes}}$, and the count of points $co \in \mathbb{N}^{n_{nodes}}$. Notice that we do not keep pointers to children, see Section 4.1.2 for reasoning. To access each layer j we have an array with the starting index of each layer $la \in \mathbb{N}^{n_{dims}}$ and an array with the dimensions that the layers represent $dims \in \mathbb{N}^{n_{dims}}$. We furthermore keep an array of the restricted dimensions $r\_dims \in \mathbb{N}^{n_{r\_dims}}$, however, for the GPU-SCY-tree in Figure 3 this is empty. To keep track of the points in the GPU-SCY-tree, we have two arrays with an entry for each point. One keeps track of the points' index in the dataset $po \in \mathbb{N}^{n_{pts}}$, and the other keeps track of which leaf-node each point is placed in $pl \in \mathbb{N}^{n_{pts}}$.

4.1.2 Restrict and merge on the GPU. When parallelizing for the GPU, we identify: (i) ways to reorder independent tasks that can be performed in parallel, (ii) similar tasks that can be performed by a warp, and (iii) ways to allocate memory as few times as possible. Restrict and merge for a SCY-tree are sequential operations where we look at one node at a time, check if it should be included, and copy all information to the temporary or final result. Running this in parallel on the GPU requires a substantial restructuring due to two things: The alternation between restrict and merge and a node's inclusion being dependent on the inclusion of either the parent or one of its children. As mentioned before, such a dependency makes the process sequential, which is not suitable for the GPU.

In Section 4, we state that all final restricted SCY-trees can be computed first in the recursion since the computation only requires the descriptors and the SCY-tree parsed to the recursion. But to parallelize the restrict and merge operation, we need several observations and restructuring that we now provide.

Allocating once. To allocate memory only once per restricted GPU-SCY-tree, we first compute which nodes and points are included in the restricted SCY-trees. This information is kept in two temporary binary arrays both initialized to **0**. One for nodes $incl \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells} \times n_{nodes}}$ with entries for each descriptor and node combination. And one for points $incl_{pts} \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells} \times n_{pts}}$ with entries for each descriptor and point combination. Here **0** and **1** represent false and true, respectively. In Figure 2, we show the restriction for descriptor (1,0). In Figure 4 we show the same restriction in GPU-SCY-tree representation, and the temporary arrays. Here the five included nodes are marked with a **1** in $incl$. Knowing which nodes and points are included allows us to compute the new indices of the nodes and points in the restricted SCY-trees. We compute the indices fornodes and points using inclusive scan (cumulative sum) of $incl$ for each descriptor. The result is kept in $idx \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$

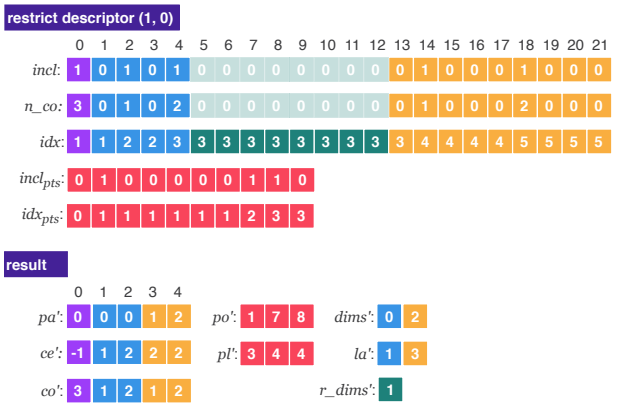


Figure 4: Restrict example before combining with merge.

and $idx_{pts} \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{pts}}$. This is used to determine where each node is placed in the resulting SCY-tree. E.g. in Figure 4, the last included node is placed at entry $4 = idx(18) - 1$. Furthermore, for each descriptor, we use the last index to allocate the needed memory for the restricted SCY-trees.In Figure 4 we need to allocate space for $5 = idx(|idx| - 1)$ nodes. After allocating memory, we copy all included nodes and points to the restricted SCY-trees. To copy, we need the new count of points in the subtrees starting at each node $n\_co \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$ which we compute along side the inclusion of each node.

Restrict is independent. We observe that the restrict operation only requires the SCY-tree parsed to the recursion and the descriptor it is restricting w.r.t.. Both the descriptor and the SCY-tree are not changed during the recursion. Therefore, the restrict operations of each recursion are completely independent of each other and all other operations. Consequently, the final result does not depend on the order of restriction, and we can parallelize the restrict operations over different thread blocks, which allows us to utilize more cores.

Restrict - similar tasks and restructuring. INSCY restricts by identifying all nodes matching a descriptor and then visiting upward and downward in the layers of the SCY-tree from there. INSCY copies all nodes on the path to the root and the subtree below the matching nodes to the new restricted SCY-tree. We take advantage of the SCY-tree being a well-balanced tree with a layer for each dimension. Observe that nodes on layers above the restricted dimension are included if any of its children is included in the restricted SCY-tree. The nodes on layers below are included if their parent is included. Because of the dependency w.r.t. inclusion between parents and children, we have a dependency between layers where we need to compute the inclusion of nodes up- and downwards in the GPU-SCY-tree starting from the restricted dimension. However, observe that computing the inclusion of each node on a layer is independent of the other nodes on that layer. Using this observation, we suggest computing the inclusion of nodes one layer at a time, making the computation of node inclusion parallel over each node on a layer. Since we keep the ordering between parents and children, we do not violate the dependency, and hence we compute the same result as INSCY.

When computing the inclusion of nodes, we have four cases, where the computation is different for each of them. One for nodes directly above the restricted dimension, one for the nodes on the remaining layers above, one for nodes directly below the restricted dimension, and one for the nodes on the remaining layers below. We handle each of the cases in their own kernel, to avoid branch-divergence that would lead to idle threads.

We compute the inclusion array *incl* in parallel with thread blocks for each descriptor $(dims(j), c)$ where j is the layer representing the restricted dimension and c is the cell number.Within each block, we process sequentially over each layer $j + k$ where $-j \leq k < n_{dims} - j$, starting from $k = 0$ and incrementing/decrementing from there. For all nodes i on a given layer we parallelize using threads.

When we compute the inclusion array *incl*, we treat normal nodes and S-connector nodes slightly differently. An S-connection is only used to enforce a merge along the restricted dimension. Therefore, we discard the S-connector path starting at the restricted dimension. Remember, we have an S-connection on the restricted dimension, when an S-connector node i has a normal node as the parent:

$$is\_S(i) := (co(i) < 0) \wedge (co(pa(i)) \geq 0). \tag{1}$$

In Figure 3, node 10 represents an S-connection since it has a negative count and its parent, node 4, has a positive count.

We can now use this when searching for nodes i matching the descriptor $(dims(j), c)$. A node i on layer j matches the descriptor $(dims(j), c)$ if its cell number matches the cell number of the descriptor $ce(i) = c$ and it is not an S-connector node starting at the restricted dimension $\neg is\_S(i)$:

$$s\_incl(j, i, c) := (ce(i) = c) \wedge (\neg is\_S(i)). \tag{2}$$

In Figure 3, for descriptor (1, 0), node 6 should be treated as a match since it is in dimension 1 and has cell number 0 and does not represent an S-connection. Node 10 also matches the descriptor, but it represents an S-connection, so it should not be treated as a match.

We wish to compute inclusion for all nodes on the layers above the restricted dimension. This requires us to look at each child of a given node. As the number of children can vary from node to node, threads in the same warp would stay idle until the other threads have visited all their children. We address this by parallelizing over all children instead and letting the children mark if their parent is included. Observe that now each thread only visits the current node and its parent, instead of a varying number of children.

Starting from layer j we compute inclusion for the nodes on layer $j-1$ just above the restricted dimension $dims(j)$. The parent $pa(i)$ of a node i is marked as included if the node i matches the descriptor $s\_incl(j, i, c)$:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, \\ &\quad la(j) \leq i < la(j+1), s\_incl(j, i, c) : \\ &\qquad incl(j, c, pa(i)) := \mathbf{1}. \end{aligned} \tag{3}$$

In Figure 4 node 2 is included since node 6 matches the descriptor.

Sequentially moving towards the root, we can now compute inclusion for nodes on layer $j - k$ where $2 \leq k < j$. The parent $pa(i)$ is now included if the node i is marked as included:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 1 \leq k < j - 1, \\ &\quad la(j-k) \leq i < la(j-k+1), incl(j, c, i) : \\ &\qquad incl(j, c, pa(i)) := \mathbf{1}. \end{aligned} \tag{4}$$

In Figure 4 the root, node 0, is included since node 2 is included.

Similarly, we include nodes on the layer $j+1$ directly below the restricted dimension $dims(j)$ if the parent matches the descriptor:

$$\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, la(j+1) \le i < la(j+2) : \\ incl(j,c,i) := s\_incl(j, pa(i), c). \tag{5}$$

In Figure 4 node 14 is included as node 6 matches the descriptor.

Moving towards the leaves, we compute inclusion for nodes on layer $j+k$ where $2 \le k < n_{dims} - j$ by checking if a node's parent is marked as included:

$$\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, 2 \le k < n_{dims} - j, \\ la(j+k) \le i < la(j+k+1) : \\ incl(j,c,i) := incl(j,c,pa(i)). \tag{6}$$

After we have computed the inclusion of the nodes on the leaf layer, we can compute which points are included. A point p is included if the leaf node where it is located $pl(p)$ is included:

$$\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, 0 \le p < n_{pts} : \\ incl_{pts}(j,c,p) := \begin{cases} incl(j,c,pl(p)) & \textbf{if } j < n_{dims} - 1 \\ ce(pl(p)) = c & \textbf{else} \end{cases} \tag{7}$$

E.g. in Figure 4 point 1 is included since the leaf node 14 where the point is placed is included. The computation is done in parallel over each descriptor as blocks and each point p as threads. We handle the case of restricting the leaf layer by directly checking if the placement node's cell number matches the descriptor.

Restrict and merge combined. INSCY alternates between restricting and merging as long as S-connections are found. The merge operation only merges restricted SCY-trees that represent subspace regions within the same subspace. Therefore, merges are independent between restricted dimensions in the same recursion. However, remember that the merge operation merges the newly restricted SCY-tree with the previous merged SCY-tree. Instead of this sequential process, we devise a strategy to perform merges and restrictions simultaneously. Implying that we avoid allocating space for the temporary restricted and merged SCY-trees, and by that, save time.

Precomputing SCY-trees to merge. Observe that in INSCY, what makes the merge process sequential, is that we do not know in advance which SCY-trees need to be merged for a given descriptor. However, this only depends on the S-connections along the restricted dimension. A merge is only necessary if there is an S-connection between two cells on the restricted dimension. We suggest precomputing which SCY-trees need to be merged for each descriptor in advance. First, check if there is an S-connection for the given descriptor, then compute from which descriptor the merging process should start. The first check for S-connections can be parallelized as follows. We define $S \in \{\mathbf{0}, \mathbf{1}\}^{n_{dims} \times n_{cells}}$, a table of whether there exists an S-connection for a given descriptor. Each entry of S is initialized to $\mathbf{0}$ and updated in parallel over each layer j as thread blocks and each node i as threads. The update entails writing $\mathbf{1}$ if the node i is the start of an S-connector path.

$$\forall 0 \le j < n_{dims}, la(j) \le i < la(j+1), is\_S(i) : \\ S(j, cells(i)) := \mathbf{1}. \tag{8}$$

We use S to compute from which descriptor each merge sequence starts. This information is saved in $M \in \mathbb{N}^{n_{dims} \times n_{cells}}$, where each entry represents a descriptor. For each entry, we compute which restricted SCY-trees should be merged, denoted by the cell number c of the descriptor associated with the first SCY-tree in that merge sequence. Remember that we start a new sequence

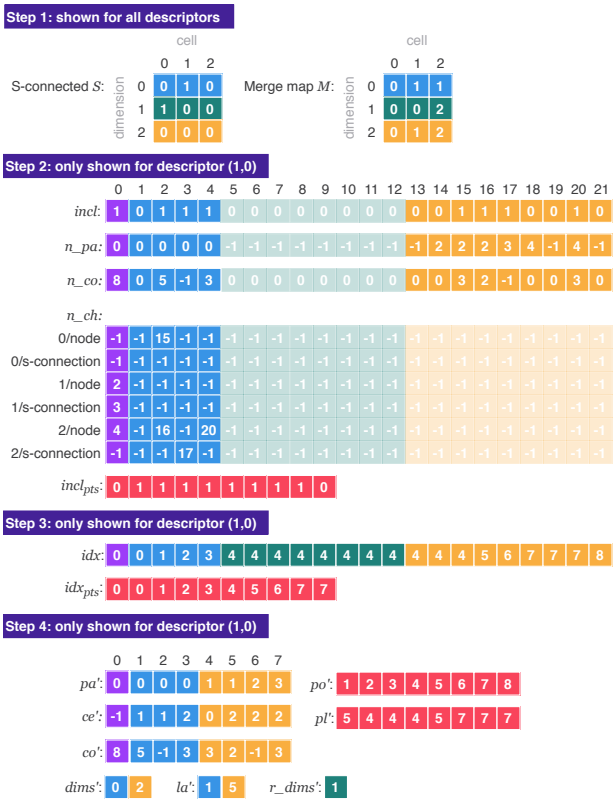


Figure 5: Restrict after combining with merge.

of merges whenever there was no S-connection from the previous cell $S(j, c-1)$. In other words, if there is an S-connection between two cells, we continue the sequence with identifier $M(j, c-1)$. If not, we start a new sequence with the identifier c.

$$\forall 0 \le j < n_{dims}, 0 \le c < n_{cells} : \\ M(j,c) := \begin{cases} M(j,c-1) & \textbf{if } (c > 0) \wedge S(j,c-1) \\ c & \textbf{else} \end{cases} \tag{9}$$

Equation 9 is parallelized over layers j but remains sequential over cell numbers c since we need to know the preceding entry $M(j, c-1)$ to compute $M(j,c)$.

The table with S-connections S and merge map M for the GPU-SCY-tree in Figure 3 are shown in Figure 5. S contains an S-connection in dimension 1, starting at cell 0. Therefore, in M, a merge sequence starts at cell 0, continuing to cell 1.

Avoiding merge sequences. The merge map M allows us to avoid the merge sequence and instead directly include nodes that would be in the final restricted SCY-tree for a given descriptor. More concretely, when checking if a node i on the restricted dimensions $d = dims(j)$ matches the descriptor, we instead look up the restricted dimension and the current node's cell number in the merge map M. We treat node i as a match if $M(j, ce(i))$ matches the cell number c of the descriptor. This changes Equation 2 into:

$$s\_incl(j,i,c) := (M(j, ce(i)) = c) \wedge (\neg is\_S(i)), \tag{10}$$

and Equation 7 into

$$\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, p < n_{pts} : \\ incl_{pts}(j,c,p) := \begin{cases} incl(j,c,pl(p)) & \textbf{if } j < n_{dims} - 1 \\ M(j,c,ce(pl(p))) = c & \textbf{else} \end{cases} \tag{11}$$

E.g. for descriptor (1, 0), we now also treat node 7 in Figure 3 and 5 as a match, since cell 1 in dimension 1 has a merge sequence starting at cell number 0.

Since nodes on the restricted dimension are not included, nodes directly below that dimension will become their grandparents' children instead. This implies that the grandparent can end up with multiple children with the same cell number. Nodes with the same parent and cell number would have been merged in INSCY and must also be merged in GPU-INSCY to ensure that INSCY and GPU-INSCY still compute the same final restricted SCY-trees. However, INSCY merges these one by one and GPU-INSCY merges them all simultaneously. In Figure 5, nodes 14 and 16 will now both be children of node 2, and they have the same cell number, so they must be merged.

Merging nodes can propagate the problem of children, with the same cell number, down towards the leaves. We merge such nodes during our new restrict phase. We keep track of nodes that need to be merged in the restricted SCY-trees by computing two things: each node's new parent $n\_pa \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes}}$ and the node's new children $n\_ch \in \mathbb{N}^{n_{dims} \times n_{cells} \times n_{nodes} \times n_{cells} \times 2}$. Examples of both arrays are shown in Figure 5. All entries of n_pa and n_ch are initialized to -1. For each descriptor, n_pa holds the new parents of all nodes. Likewise for n_ch, except that we make room for all possible children by $n_{cells} \times 2$. A node can have two types of children: normal or S-connector nodes. For both types, we can have a node for each cell. To look up the type of a node we use:

$$S\_idx(i) := \begin{cases} \mathbf{0} & \textbf{if } co(i) \geq 0 \\ \mathbf{1} & \textbf{else} \end{cases} \tag{12}$$

Merge representatives. When merging nodes in the SCY-tree, we pick one of the nodes to be the representative, which is the node that will actually be included in the final restricted SCY-tree. We will lookup the representative node $rep(j, c, i)$ by

$$rep(j, c, i) := n\_ch(j, c, n\_pa(j, c, i), ce(i), S\_idx(i)).$$

If a node should be represented in the final restricted SCY-tree we say that it is fused into that SCY-tree. We call it fused if it is either merged or included in the SCY-tree. If a node is merged into the SCY-tree, the count of points and children is added to the representative node. In Figure 5, nodes 14 and 16 should be fused, but only node 16 is included as the representative.

We assign a new parent to all nodes that are fused into the final restricted SCY-tree. This implies that iff n_pa has a value that is not -1, the associated node has been fused into the final restricted SCY-tree. Notice that we can use $n\_pa(j, c, i) \geq 0$ to check if the parent has been fused instead of just checking if it has been included $incl(j, c, i)$.

When identifying the new parent of a node i, below the restricted dimension, we look up which node the old parent has been merged into. This will be one of the children of the new grandparent of node i, which is identified as the representative node for the parent:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 2 \leq k < n_{dims} - j, \\ &\quad la(j+k) \leq i < la(j+k+1), n\_pa(j, c, pa(i)) \geq 0 : \\ &\qquad n\_pa(j, c, i) := rep(j, c, pa(i)). \end{aligned} \tag{13}$$

When computing the new parent for nodes just below the restricted dimension, we need to skip the nodes on the restricted dimension, since the restricted layer is removed from the result. However, for a node above the restricted dimension, there are no changes. Therefore, no merge of nodes can occur, and we do not need to check which child has been picked:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, \\ &\quad la(j+1) \leq i < la(j+2), s\_incl(j, pa(i), c) : \\ &\qquad n\_pa(j, c, i) := pa(pa(i)). \end{aligned} \tag{14}$$

E.g., the parent of node 14 is node 6, and the parent of node 6 is node 2. Therefore, the new parent of node 14 is node 2.

For all nodes above the restricted dimension, we do not change the child-parent relationship, and they can be copied in parallel.

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 1 \leq k < j, \\ &\quad la(j-k) \leq i < la(j-k+1), I(j, c, i) : \\ &\qquad n\_pa(j, c, i) := pa(i), \\ &\qquad n\_ch(j, c, pa(i), ce(i), S\_idx(i)) := i. \end{aligned} \tag{15}$$

Below the restricted dimension, we need to decide which of the merged nodes is the representative. It is not important which of the nodes is picked, but all threads involved in the merge must agree on just one node. We do this by letting each node i, that is fused, write its id as the representative, i.e., the new child:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 1 \leq k < n_{dims} - j, \\ &\quad la(j+k) \leq i < la(j+k+1), n\_pa(j, c, i) \geq 0 : \\ &\qquad rep(j, c, i) := i. \end{aligned} \tag{16}$$

We synchronize such that all threads see the same node id, and only include that node as the new child. E.g., in Figure 5 both node 14 and 16 would vote for themselves as the representative. In our example, node 16 was the last to write. Therefore, node 16 becomes the representative. This expands Equation 5 into:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, la(j+1) \leq i < la(j+2) : \\ &\quad incl(j, c, i) := s\_incl(j, pa(i), c) \wedge (rep(j, c, i) = i), \end{aligned} \tag{17}$$

and Equation 6 into:

$$\begin{aligned} &\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, 2 \leq k < n_{dims} - j, \\ &\quad la(j+k) \leq i < la(j+k+1) : \\ &\qquad incl(j, c, i) := (n\_pa(j, c, i) \geq 0) \wedge (rep(j, c, i) = i). \end{aligned} \tag{18}$$

For a point, the placement can change since nodes are merged. Therefore, we check if the node where the point is placed is fused into the final restricted SCY-tree. This is the case if the node has been assigned a new parent. Equation 11 changes into:

$$\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, p < n_{pts} :$$

$$incl_{pts}(j, c, p) := \begin{cases} n\_pa(j, c, pl(p)) \geq 0 & \textbf{if } j < n_{dims} - 1 \\ M(j, c, ce(pl(p))) = c & \textbf{else} \end{cases} \tag{19}$$

Accumulating count. Now that we know which nodes are fused into the SCY-tree, we can accumulate the count of points in the subtree of each node i. For nodes on the same layer, the entry in n_co might be incremented by different threads. Therefore, we need to use atomic addition, implying that threads handling nodes on the same layer must be in the same thread block. For the layer just above the restricted dimension, we sum the old count of all children that are normal nodes and fused. If the parent is included and an S-connector node, we set the count to -1:

$$\forall 0 \leq j < n_{dims}, 0 \leq c < n_{cells}, la(j) \leq i < la(j+1), s\_incl(j, i, c) :$$

$$n\_co(j, c, pa(i)) := \begin{cases} n\_co(j, c, pa(i)) + co(i) & \textbf{if } co(i) \geq 0 \\ -1 & \textbf{if } co(pa(i)) < 0 \end{cases} \tag{20}$$

For the nodes on the remaining layers above the restricted dimension, we iteratively sum the new count of points of the children:

$$\begin{aligned}&\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, 1 \le k < j-1,\\ &\quad la(j-k) \le i < la(j-k+1), incl(j,c,i):\\ &\qquad n\_co(j,c,n\_pa(j,c,i)) :=\\ &\qquad\quad + \begin{cases} n\_co(j,c,pa(i)) + n\_co(j,c,i) & \textbf{if } co(i) \ge 0\\ -1 & \textbf{if } co(pa(i)) < 0 \end{cases}\end{aligned} \tag{21}$$

For all layers below the restricted dimension, the new count is a sum of the old counts of all fused nodes:

$$\begin{aligned}&\forall 0 \le j < n_{dims}, 0 \le c < n_{cells}, 1 \le k < n_{dims} - j,\\ &\quad la(j+k) \le i < la(j+k+1), n\_pa(i) \ge 0:\\ &\qquad n\_co(j,c,rep(j,c,i)) := \begin{cases} -1 \textbf{ if } co(i) < 0\\ n\_co(j,c,rep(j,c,i)) + co(i) \textbf{ else} \end{cases}\end{aligned} \tag{22}$$

Overview of restrict and merge operations. To summarize, the restricting and merging for all descriptors is done by

- Initialization: Each entry of $incl$, $incl_{pts}$, idx, idx_{pts}, and n_co is initialized to 0. Each entry of n_ch and n_pa is initialized to -1.
- Step 1: Compute for which descriptors the associated SCY-trees will be merged using two kernels; one that checks for each descriptor if there is an S-connection, using Equation 8, and one that uses this information to compute which SCY-trees will be merged, using Equation 9.
- Step 2: Compute which nodes are included in the final restricted and merged SCY-trees, and accumulate the count of points in the subtrees. We compute the inclusion in the restriction using five kernels. First, directly above the restricted dimension we use Equations 3, 15, and 20, second, for the remaining layers above we use Equations 4, 15, and 21, third, directly below we use Equations 17, 14, 16, and 22, fourth, for the remainder below we use Equations 18, 13, 16, and 22, and at last, we compute inclusion of points by checking if the leaf-node where the point is placed is included using Equation 19.
- Step 3: We now know which nodes and points are included in the final restricted SCY-trees. We do an inclusive scan and decrement each entry with 1 to compute the new indices for nodes idx and points idx_{pts}. This is also used to allocate the arrays for all final restricted SCY-trees.
- Step 4: All needed information has been precomputed, and we now copy all nodes, points, dimensions, and restricted dimensions to the final restricted SCY-trees. Each copy is independent and can be done completely in parallel.

4.2 Density-based clustering on the GPU

In this section, we discuss how to find the subspace clusters for all points in each SCY-tree. For each subspace region, the clustering process of INSCY is similar to that of DBSCAN [12]. The main difference is that INSCY supports different density measures and that clustering is done in a subspace projection. DBSCAN, and other density-based clustering methods, find clusters by expanding chains of density-connected points. This is a sequential process that we would like to replace with a parallelized process.

As discussed in related work, G-DBSCAN [5] is a competitive parallelization of full-space DBSCAN with rectangle kernel for density assessment. To support INSCY subspace clustering and further improve runtime performance, we introduce three major algorithmic solutions: supporting a different unbiased, i.e., subspace-dependent density-measure, reduced neighborhood searches, and expanding several clusters at once.

Precomputing the neighborhoods. To compute the neighborhood without allocating worst-case sizes, G-DBSCAN first computes the neighborhoods' size, then allocates space, and at last populates the neighborhoods with the neighboring points. For GPU-INSCY, the neighborhood of each point in all SCY-trees can be computed independently of other points and can therefore be computed in parallel over different thread blocks.

GPU-INSCY additionally takes advantage of already having computed the neighborhoods in the lower-dimensional subspace projections of the current subspace. Since adding a dimension to a subspace only increases the distance between points, previous neighborhoods can be used to bound the search for neighbors effectively. We demonstrate that this is an efficient strategy in the experiments, see Section 5.

Collecting the clusters. Using the precomputed neighborhoods, G-DBSCAN proceeds as follows. While there are still unclustered points, pick a random point to expand a cluster from. While that cluster is still being expanded, look at all points in parallel. If a point has just been added to the cluster, add its neighbors that have not yet been clustered to the current cluster. Since G-DBSCAN run in parallel for all points, but only a few points actually expand a single cluster each iteration, many threads are left idle. We suggest instead that a point adds itself to a cluster. Furthermore, we expand all clusters simultaneously for each point p in parallel as threads and over each descriptor in parallel as blocks. We precompute for each point if it is dense and only perform the following for dense points. For each descriptor, let $C \in \mathbb{N}^{n_{pts}}$ be clustering labels for each point p in the SCY-tree associated with that descriptor. Start by assigning all points to a singleton cluster, letting the cluster id be the point id, $C(p) := p$. While any cluster is still being expanded, look at all points in parallel. If the point p can reach a cluster with a lower cluster-id through its neighborhood, add the current point to that cluster $C(p) := \min_{q \in N_\varepsilon(p) \cup \{p\}} C(q)$. Between each iteration, we synchronize such that all threads know if any cluster has been expanded. For each iteration we check for all points if they can be expanded, thus we ensure that all density connected clusters have been found.

Clustering of each subspace region (SCY-tree) is independent of each other since the subspace regions are not S-connected, meaning that no density-connected clusters can span multiple subspace regions. Therefore, since no communication is needed, we can compute the clustering in parallel for each SCY-tree using different thread blocks. However, since we want to perform all clusterings in parallel and each SCY-tree must have been pruned first, we can only perform clusterings in parallel at the end.

4.3 Pruning on the GPU

As previously mentioned, we parallelize both pruning phases. In the interest of space, we keep the discussion brief as it follows the same approach as for restricting and merging the GPU-SCY-trees.

When pruning the recursion, we compute in parallel for each point if it is weak-dense. If it is not, mark it as not-included and propagate the count up in the SCY-tree layer by layer. We also parallelize the propagation over all nodes on a layer. If the count in the root is below min_C, then we do not continue with the recursion for this SCY-tree.

Pruning for redundancy is done as follows. For each superspace of the current subspace, we execute three kernels: Find the size of each cluster, find all clusters that overlap with points in the current SCY-tree, and find the smallest cluster that overlaps with the points in the current SCY-tree. Update the largest smallest cluster *max_min_cluster* that overlaps with the current SCY-tree. If the number of points in the SCY-tree scaled by the parameter r is smaller than *max_min_cluster*, we do not perform clustering for this SCY-tree because it can only contain redundant clusters.

4.4 Trading off speed for memory usage

Each recursive call of GPU-INSCY is parallelized over all descriptors simultaneously. This requires that we keep all final restricted SCY-trees, neighborhoods, and clusters in memory for all descriptors. However, memory on the GPU is limited, putting a bound on how large inputs we can process in parallel. There is, therefore, a natural trade-off between memory usage and how many descriptors we efficiently parallelize over simultaneously. To support efficient processing of larger inputs, we devise a version of GPU-INSCY called GPU-INSCY-memory that iterates over subsets of descriptors that we then parallelize over. We study this trade-off experimentally in Section 5.

5 EXPERIMENTS

5.1 Experimental setup

We conduct experiments for comparison of GPU-INSCY with INSCY on synthetic and on real-world data, and study impact of parameters on a workstation with Intel Core i7-9750HF CPU 2.60GHz × 12 cores, 16 GB RAM, GeForce GTX 1660 TI 6 GB dedicated RAM. The large scale experiments in Section 5.4 are executed on NVIDIA TITAN V 12 GB dedicated RAM, Intel Core E5-2687W 3.100GHz × 10 cores, 400 GB RAM.

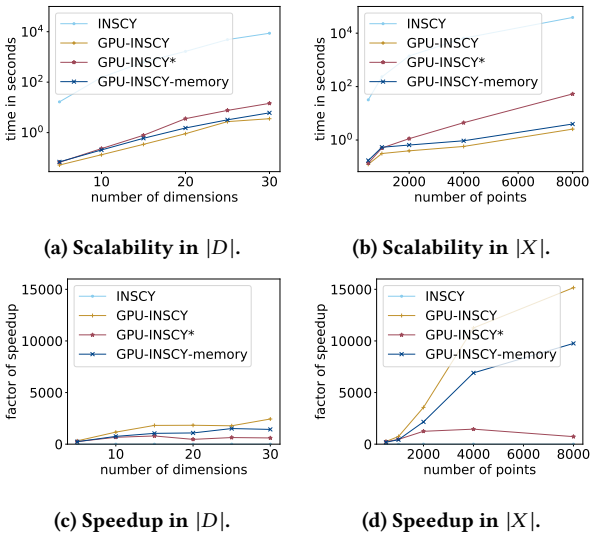
We use a search-tree for efficient neighborhood search in INSCY, which provides a large speedup and makes it a fairer comparison. We have experimentally validated that GPU-INSCY and INSCY produce identical subspace clusterings. Plots and further details have been omitted due to the space limit. We provide our source code at: https://github.com/jakobrj/GPU_INSCY.

5.2 Comparison with INSCY.

For subspace clustering, the dimensionality and size of the dataset are dominating factors regarding runtime. Especially dimensionality since, as the number of dimensions increases, the number of possible subspaces increases exponentially.

To compare INSCY and GPU-INSCY and the impact of input data, we use the data generator provided by [1] to generate synthetic datasets with dense clusters in arbitrary subspaces that may overlap and have a small percentage of noise. As in [7], we generate different datasets with four hidden subspace clusters. All runtimes are averages of three runs on datasets with the same generator settings. All dataset have been min/max-normalized. The default parameters for INSCY and GPU-INSCY in these experiments are $F = 1$, $R = 1$,$\mu = 8$, $\varepsilon = 0.01$, $n_{cells} = 4$, and min_C is set to 5% of the data points.

To analyze components of our algorithm, we also test GPU-INSCY* and GPU-INSCY-memory. GPU-INSCY* is a version of GPU-INSCY that does not bound the neighborhood search, so that we can study the effect of bounding the neighborhood search. GPU-INSCY-memory is described in Section 4.4. For our experiments we group the descriptors by the dimensions such that each iteration of the recursions is only parallel over the cells.



(a) Scalability in $|D|$. (b) Scalability in $|X|$.

(c) Speedup in $|D|$. (d) Speedup in $|X|$.

Figure 6: Scalability in size and dimensionality

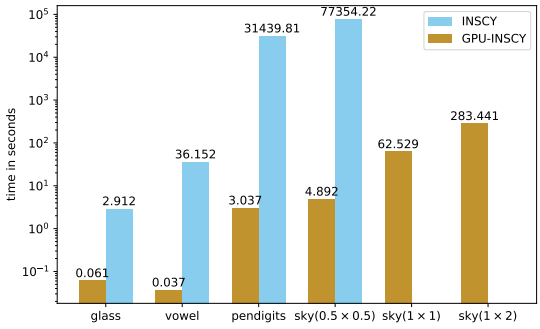


Figure 7: Real world data; INSCY aborted after 24 hours

Comparison of INSCY and GPU-INSCY. In Figure 6a, the running time for INSCY is decent for lower dimensions but increases rapidly for higher dimensions. GPU-INSCY reduces the running time to a point where it is faster to find subspace clusters for 25 dimensions using GPU-INSCY than finding subspace clusters for two dimensions using INSCY. In fact, in Figure 6c, the speedup of GPU-INSCY relative to INSCY keeps increasing. For 30 dimensions we achieve a factor of speedup of more than 2000×. A similar effect is observed for increasing the number of points. In Figure 6b, INSCYs runtime again increases faster than for GPU-INSCY. In Figure 6d, we see that the speedup becomes a factor of several thousand. This speedup is much higher than expected for the relatively low number of 1536 cores on our GPU.

Comparison of versions of GPU-INSCY. As mentioned in Section 4.2, we attribute this dramatic speedup to our bounding of the neighborhood searches. This effect is also clear in Figure 6c and 6d, where we see that GPU-INSCY* achieves a 500-1000× speedup, corresponding to a good use of the cores, and GPU-INSCY achieves a substantially larger speedup of up to 14'000× obtained by our improved neighborhood search. GPU-INSCY-memory allows us to run on larger datasets, with only a slight reduction of factor 2 in speedup, which is a reasonable trade-off.

5.3 Real world datasets

We also demonstrate GPU-INSCY speedups for real-world datasets. We report runtimes on the three datasets (glass, vowel, pendigits) [23] also studied in [7, 8]. The glass dataset $X_{glass} \in \mathbb{R}^{214\times9}$, vowel $X_{vowel} \in \mathbb{R}^{990\times10}$, and pendigits $X_{pendigits} \in \mathbb{R}^{7494\times16}$. Furthermore, we also evaluate on a more sizable higher dimensional real world data set, part of the SkyServer dataset[27] that contains measurements of objects in the sky, e.g., stars and galaxies. We select three different areas of size 0.5×0.5, 1×1, and 1×2, measured in spherical coordinates (RA/Dec): $X_{sky(0.5\times0.5)} \in \mathbb{R}^{7253\times17}$, $X_{sky(1\times1)} \in \mathbb{R}^{29627\times17}$, and $X_{sky(1\times2)} \in \mathbb{R}^{59285\times17}$. Experiments are aborted if they run for more than 24 hours, as INSCY does for larger setups. In Figure 7, we see that we obtain high speedups on all datasets, but much higher for larger datasets up to 15′000× speedup.

5.4 Effect of parameters

In this section, we study the effect of parameters for the density criterion, ε, μ, F and the model parameter n_{cells}.

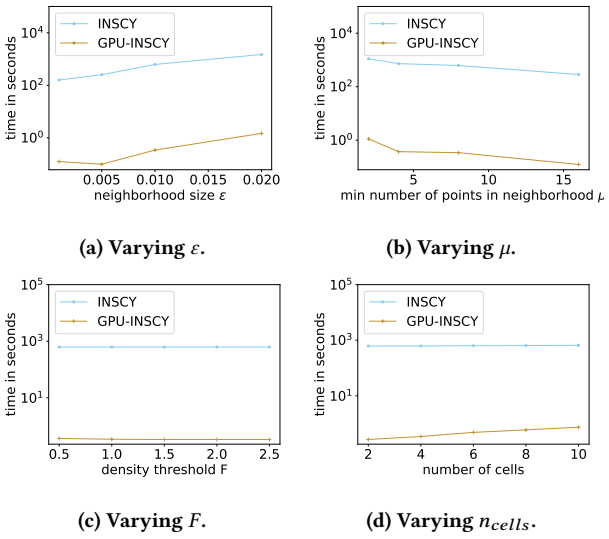
In particular, the parameters for the density criterion are expected to impact the running time. Especially the neighborhood radius ε is interesting since GPU-INSCY uses a strategy for reducing the neighborhood search that INSCY does not employ. The bigger the part of the subspace region that the neighborhood radius covers, the less we save by reducing the search area for the neighborhoods. Therefore, we expect that GPU-INSCY will obtain the greatest speedup for smaller values of ε. In Figure 8a, we study the range of ε between 0 and 0.02, and see that this is the case, but that the speedup remains large for the entire range.

The minimum number of points in the neighborhood μ and density threshold F only affect the number of points that are dense and weak-dense. The fewer points that are dense or weak-dense, the fewer points INSCY and GPU-INSCY need to process. As this is the same fraction of points for both INSCY and GPU-INSCY, we, therefore, expect to see a similar reduction in time for both algorithms. For μ, we study the range between 2 and 16, as this parameter is intended as a cut-off for avoiding tiny subspace clusters in very high-dimensional subspace projections (called pseudodense in INSCY). The factor F that governs the extent to which expected density is exceeded is evaluated in the range between 0.5 and 2.5. A value of 0.5 implies that we only expect a point to be half as dense as the expected density, which is a very low criterion, and 2.5 is more than twice the expected density, which is rather high. In Figure 8b and 8c, we see that the speedup for the density parameters remains stable for both criteria. As expected, we see that the running time decreases equally for both INSCY and GPU-INSCY as μ increases.

The parameter number of cells n_{cells} does not change the result, but only how we partition the subspace into cells and regions. We can, therefore, pick whichever number of cells INSCY or GPU-INSCY perform the best at. In Figure 8d, we study a range between 2 and 10 cells. Here both INSCY and GPU-INSCY perform best at a lower number of cells, especially GPU-INSCY.

5.5 Scalability and different distributions

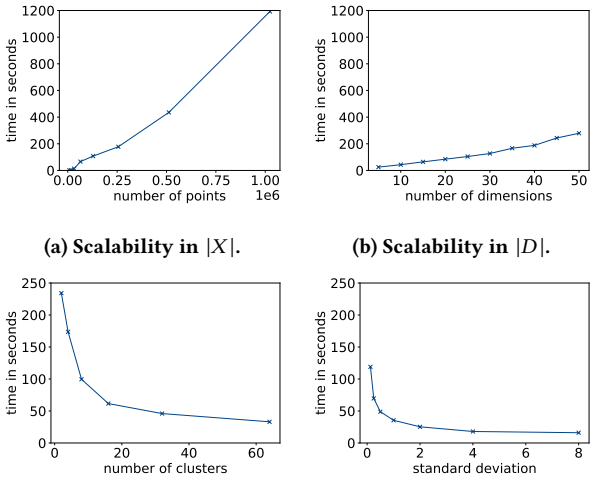
We evaluate scalability and different data distributions for GPU-INSCY alone. The running time of INSCY quickly becomes too high, e.g., more than 10 hours for 8000 points and 15 dimensions, which makes experiments for large inputs infeasible. In this section, we use GPU-INSCY-memory.



(a) Varying ε. (b) Varying μ.

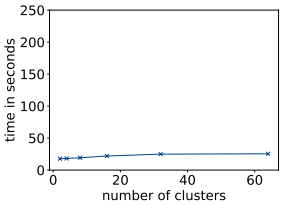
(c) Varying F. (d) Varying n_{cells}.

Figure 8: Runtimes for different parameter values



(a) Scalability in $|X|$. (b) Scalability in $|D|$.

(c) Scalability in number of clusters. (d) Scalability in cluster spread (standard deviation).



(e) Scalability in number of clusters with stable density.

Figure 9: Runtimes for scalability experiments

To test various distributions, we use the generator provided by [9], but modify it to generate clusters in random subspaces and not just the first k dimensions. The default settings used for the dataset generator are 64‘000 points with 4000 points for each cluster, except 1%, which is uniformly distributed noise. The dataset values range from −100 to 100, and the full space consists of 15 dimensions. Each cluster is normally distributed with a

standard deviation of 0.3 in a random 3-dimensional subspace. All datasets have been min/max-normalized. The default parameters for GPU-INSCY in these experiments are $F = 0.1$, $R = 1$, $\mu = 1$, $\varepsilon = 0.0003$, $n_{cells} = 4$, and $min_C = 500$ points.

Scalability. Figure 9a shows runtimes with increasing dataset size $|X|$ up to 1‘024‘000. The figure shows that GPU-INSCY performs subspace clustering on 1‘024‘000 points in less that 20 minutes. We also run experiments for increasing dimensionality $|D|$ up to 50, as shown in Figure 9b. GPU-INSCY can perform subspace clustering in 50 dimensions (and on 64‘000 points) in less than 6 minutes.

Data distribution. We evaluate performance on data with different distributions using the same setting as for scalability. In Figure 9c, we increase the number of clusters, keeping cluster distribution (standard deviation) and total number of points fixed. As we can see, large numbers of clusters further reduce the runtime of GPU-INSCY, as it finds fewer points in each neighborhood when the number of points per cluster decreases. In Figure 9d, we increase the spread of clusters (standard deviation). Again, the runtime of GPU-INSCY further improves, as neighborhoods are again less populated. Finally, we conduct an experiment with stable density. As the number of clusters is doubled, we increase cluster density accordingly by halving standard deviation. As Figure 9e confirms, similar density results in stable runtime when scaling number of clusters.

To summarize, the trend is that a lower density implies fewer points in each neighborhood and, therefore, a lower runtime. This means that GPU-INSCY scales particularly well for large numbers of clusters and clusters that are spread.

Overall, GPU-INSCY outperforms INSCY by two-four orders of magnitude with respect to runtimes for all tested settings. Even on our small GPU, we measure the running time in seconds instead of hours for smaller datasets (< 10‘000 points and 15 dimensions) and minutes instead of days for larger datasets.

6 CONCLUSION

In this paper, we propose GPU-INSCY, a novel GPU-parallel algorithm for dimensionality-unbiased density-based subspace clustering, following INSCY. GPU-INSCY outperforms INSCY by several orders of magnitude. To achieve this, we utilize GPU cores by restructuring both the algorithmic processing and the data structure SCY-tree used in INSCY to fit the GPU computational model. Furthermore, GPU-INSCY proposes a further reduction of the time spent on neighborhood searches. Our experiments show that GPU-INSCY scales well w.r.t. dimensionality and size of the dataset, and compared to INSCY, the gap even continues to grow with the scale of data.

ACKNOWLEDGMENTS

This work was supported by Independent Research Fund Denmark.

REFERENCES

[1] Andrew Adinetz, Jiri Kraus, Jan Meinke, and Dirk Pleiter. 2013. GPUMAFIA: Efficient subspace clustering with MAFIA on GPUs. In *European Conference on Parallel Processing*. Springer, 838–849.

[2] Charu C Aggarwal, Joel L Wolf, Philip S Yu, Cecilia Procopiuc, and Jong Soo Park. 1999. Fast algorithms for projected clustering. *ACM SIGMoD Record* 28, 2 (1999), 61–72.

[3] Charu C Aggarwal and Philip S Yu. 2000. Finding generalized projected clusters in high dimensional spaces. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 70–81.

[4] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. 1998. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 94–105.

[5] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. 2013. G-dbscan: A gpu accelerated algorithm for density-based clustering. *Procedia Computer Science* 18 (2013), 369–378.

[6] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. 2007. DUSC: Dimensionality unbiased subspace clustering. In *seventh IEEE international conference on data mining (ICDM 2007)*. IEEE, 409–414.

[7] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. 2008. EDSC: efficient density-based subspace clustering. In *Proceedings of the 17th ACM conference on Information and knowledge management*. 1093–1102.

[8] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. 2008. INSCY: Indexing subspace clusters with in-process-removal of redundancy. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 719–724.

[9] Anna Beer, Nadine Sarah Schüler, and Thomas Seidl. 2019. A Generator for Subspace Clusters.. In *LWDA*. 69–73.

[10] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. 2009. Density-based clustering using graphics processors. In *Proceedings of the 18th ACM conference on Information and knowledge management*. 661–670.

[11] Chun-Hung Cheng, Ada Waichee Fu, and Yi Zhang. 1999. Entropy-based subspace clustering for mining numerical data. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. 84–93.

[12] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[13] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H Campbell. 2008. A Parallel Implementation of K-Means Clustering on GPUs.. In *Pdpta*, Vol. 13. 212–312.

[14] Sanjay Goil, Harsha Nagesh, and Alok Choudhary. 1999. MAFIA: Efficient and scalable subspace clustering for very large data sets. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Vol. 443. ACM, 452.

[15] Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. 2004. Density-connected subspace clustering for high-dimensional data. In *Proceedings of the 2004 SIAM international conference on data mining*. SIAM, 246–256.

[16] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. 2009. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 3, 1 (2009), 1–58.

[17] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. 2013. Speeding up k-means algorithm by gpus. *J. Comput. System Sci.* 79, 2 (2013), 216–229.

[18] Woong-Kee Loh, Yang-Sae Moon, and Young-Ho Park. 2014. Erratum: Fast Density-Based Clustering Using Graphics Processing Units [IEICE Transactions on Information and Systems Vol. E97. D (2014), No. 5 pp. 1349-1352]. *IEICE TRANSACTIONS on Information and Systems* 97, 7 (2014), 1947–1951.

[19] Woong-Kee Loh and Hwanjo Yu. 2015. Fast density-based clustering through dataset partition using graphics processing units. *Information Sciences* 308 (2015), 94–112.

[20] Gabriela Moise and Jörg Sander. 2008. Finding non-redundant, statistically significant regions in high dimensional data: a novel approach to projected and subspace clustering. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 533–541.

[21] Emmanuel Müller, Stephan Günnemann, Ira Assent, and Thomas Seidl. 2009. Evaluating clustering in subspace projections of high dimensional data. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1270–1281.

[22] Hamza Mustafa, Eleazar Leal, and Le Gruenwald. 2019. An Experimental Comparison of GPU Techniques for DBSCAN Clustering. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 3701–3710.

[23] David J Newman, SCLB Hettich, Cason L Blake, and Christopher J Merz. 1998. UCI repository of machine learning databases, 1998.

[24] Lance Parsons, Ehtesham Haque, and Huan Liu. 2004. Subspace clustering for high dimensional data: a review. *Acm Sigkdd Explorations Newsletter* 6, 1 (2004), 90–105.

[25] Karlton Sequeira and Mohammed Zaki. 2005. SCHISM: a new approach to interesting subspace mining. *International Journal of Business Intelligence and Data Mining* 1, 2 (2005), 137–160.

[26] Kelvin Sim, Vivekanand Gopalkrishnan, Arthur Zimek, and Gao Cong. 2013. A survey on enhanced subspace clustering. *Data mining and knowledge discovery* 26, 2 (2013), 332–397.

[27] Alexander S Szalay, Jim Gray, Ani R Thakar, Peter Z Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. 2002. The SDSS skyserver: public access to the sloan digital sky server data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 570–581.

[28] Rajeev J Thapa, Christian Trefftz, and Greg Wolffe. 2010. Memory-efficient implementation of a graphics processor-based cluster detection algorithm for large spatial databases. In *2010 IEEE International Conference on Electro/Information Technology*. IEEE, 1–5.

[29] Kyoung-Gu Woo, Jeong-Hoon Lee, Myoung-Ho Kim, and Yoon-Joon Lee. 2004. FINDIT: a fast and intelligent subspace clustering algorithm using dimension voting. *Information and Software Technology* 46, 4 (2004), 255–271.

JIT happens: Transactional Graph Processing in Persistent Memory meets Just-In-Time Compilation

Muhammad Attahir Jibril
TU Ilmenau
Germany
muhammad-attahir.jibril@tu-ilmenau.de

Alexander Baumstark
TU Ilmenau
Germany
alexander.baumstark@tu-ilmenau.de

Philipp Götze
TU Ilmenau
Germany
philipp.goetze@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau
Germany
kus@tu-ilmenau.de

ABSTRACT

Graph databases are used for different applications like analyzing large networks, representing and querying knowledge graphs, and managing master data and complex data structures. Besides graph analytics, the transactional processing of concurrent updates and queries represents a challenging data management task. In this paper, we investigate the usage of persistent memory as a very promising technology for graph processing. We present a novel architecture for transactional processing of queries and updates on a property graph model that exploits and addresses the specific characteristics of persistent memory by hybrid storage and memory management as well as a just-in-time query compilation approach. Our experimental evaluation on interactive short read and update query workloads show that PMem-based systems that are well-designed to exploit PMem characteristics outperform traditional disk-based systems significantly and have only a small overhead compared to DRAM-only systems. Moreover, the evaluation shows that JIT compilation brings performance benefits especially when an adaptive compilation approach is leveraged to hide the overhead of compilation as well as the latency of PMem.

1 INTRODUCTION

Graph databases represent an important class of NoSQL systems with numerous flavors, including systems for analyzing large graphs, systems for querying knowledge bases, and systems supporting updates on graphs and navigational queries. They are designed for different graph data models ranging from RDF triples to property graph models, as well as different processing models from database query processing to approaches like the bulk synchronous parallel (BSP) model.

The numerous available systems mainly adopt the typical architectures of database systems, i.e., traditional disk-based architecture, in-memory architecture or scalable, distributed solutions. Graph data are either stored in disk-based data structures and loaded into memory for processing or kept directly in in-memory structures (without requiring to load data during startup) while using techniques like logging to allow for persistent updates.

In this work, we present a novel architecture for graph databases based on persistent memory (PMem). PMem – also known as non-volatile memory (NVM) or storage-class memory (SCM) – is one of the most promising trends in hardware development which have the potential to hugely impact database system architectures. Characteristics such as byte-addressability, read latency close to DRAM but with read-write asymmetry, and inherent persistence open up new opportunities for database systems. Specifically, Intel's Optane DC Persistent Memory Modules (DCPMMs) are already available on the market and supported by the Persistent Memory Development Kit (PMDK) [17]. Several studies, as well as our experiments, have identified the following characteristics of this technology (we elaborate these in more detail in Section 3):

(C1) PMem has a higher latency and lower bandwidth than DRAM.
(C2) Reads and writes on PMem behave asymmetrically.
(C3) DCPMMs internally work on 256-byte blocks.
(C4) Failure atomicity is only guaranteed for 8-byte aligned writes.

The focus of our work is an architecture for hybrid transactional/analytical processing (HTAP) on a property graph model. Transaction support covers insert/update/delete operations on nodes, relationships, and their properties with ACID guarantees. Furthermore, we support Cypher-like navigational queries. In this paper, we particularly focus on data structures and techniques for query and transaction processing in graph databases exploiting PMem and addressing the characteristics (C1)-(C4) mentioned above. Although we aim for HTAP, we do not consider graph analytics in this paper yet. Exploiting PMem for graph analytics is discussed by other researchers, e.g., in [13]. Our contributions are as follows:

- We present the architecture of an HTAP graph engine with storage structures designed for PMem, primarily taking (C1)-(C3) into account.
- We discuss the implementation of a timestamp ordering-based multiversion concurrency control (MVTO) protocol optimized for PMem addressing (C4).
- We describe our just-in-time (JIT) query compilation approach for compiling graph queries into machine code to hide the higher latency of PMem as described in (C1).

Thus, the novelty of our work lies in the design, adaptation as well as evaluation of transaction and query processing techniques to leverage the idiosyncrasies of persistent memory for graph databases.

2 RELATED WORK

Several of the approaches presented in this paper are based on insights from previous work. In particular, the lessons learned regarding the new concepts of data structures for PMem had a

major impact on the design decisions for our graph engine. There is, to our knowledge, no transactional graph system or JIT query compilation approach targeting persistent memory, yet. Hence, we approach the subject from three directions: general graph management, PMem-aware data structures and storage engines as well as query compilation.

Graph Management. For graph data management, numerous data models and systems have been proposed in the past. Among the several database models for graph data [3], RDF for the Semantic Web and property graph models are the most prominent. On top of these, query languages like SPARQL for RDF triple data, diverse SQL dialects, and dedicated languages like Cypher[1] and Gremlin [36] have been developed. The SQL standardization committee is currently working on standardizing the graph query language GQL.

Depending on the supported data model and query language, graph database systems are either special-purpose systems such as triple stores for RDF like Virtuoso, native stores for property graphs e.g., Neo4j, relationally-backed approaches such as DB2RDF [7] and EmptyHeaded [1]; or extensions of SQL systems like Grail [11] and SAP HANA [37]. Here, standard DBMS implementation techniques are used for data storage, indexing, transaction management, and query processing. Particularly, traversal operations [32], as well as support for graph analytics [29, 38], play an important role. However, only very few approaches try to support HTAP workloads (TigerGraph, Neo4j) and to our best knowledge, no established graph system is utilizing PMem yet [6].

Recently, however, Gill et al. [13] investigated the application of DCPMMs in Memory mode for running graph analytics. They evaluated large scale data sets on existing graph frameworks and demonstrated that their NUMA-aware algorithms on cheaper single machine setups with DCPMMs can outperform more expensive DRAM-only cluster setups. With Sage [9], the authors have shown that the AppDirect mode of DCPMMs in combination with sophisticated algorithms can even achieve a better performance than an unmodified in-memory graph database used on PMem in Memory mode. They especially address the asymmetry of PMem by introducing the parallel semi-asymmetric model. Here, the entire graph is stored as a read-only copy in PMem and a smaller mutable part in DRAM. A volatile auxiliary structure keeps track of deleted edges for graph filtering. Since the focus is on parallel analytical queries, we assume that no transactional updates are possible. In this paper, we want to make the appropriate contribution in this regard.

PMem-aware Storage Designs. Researchers recently started adapting existing data structures to PMem. This includes several variants of the B$^+$-Tree [8, 43], hybrid variants like the FPTree [31], the LB$^+$-Tree [27], DPTree [49], and HiKV [47], as well as LSM-Tree variations [19]. There are also latch-free B$^+$-Tree variants targeting modern hardware, such as the Bw-Tree [26, 45] and the BzTree [4]. While the former addresses multi-core systems with flash storage, the BzTree is explicitly designed for PMem. Apart from the individual data structures, some approaches for PMem-based storage engines have been proposed. SOFORT [30] is a columnar transactional storage engine leveraging PMem by minimizing logging and updating data in place, aiming for mixed OLAP and OLTP workloads. Peloton [33] is another relational DBMS engine already considering PMem by applying write-behind logging [5]. The basic idea is to write and flush all changed entries in-place to PMem during commit. A more recent proposal is the key-value store RStore [25]. It opts for a log-structured design with an index. It utilizes linked and fix-sized append-only blocks in PMem. Once a block is full, it is considered immutable and indexed in a volatile tree which is rebuilt during recovery. Additionally, RStore employs partitions that are owned by only one thread at a time, each having its own log to parallelize recovery.

JIT Query Compilation. Similarly, there are numerous works on query compilation techniques. Neumann [28] presented a query compiler architecture using the LLVM framework[2] to generate and compile code for queries in the HyPer database. Based on this, Kohn et al. [21] proposed an approach to mask the compilation time by compiling the query in the background while interpreting it. They further improve the efficiency by using different execution modes depending on the query type. There are also works that try to provide a lightweight approach, apart from LLVM. An alternative approach, LegoBase, provides a query compiler that generates high-level code in multiple steps, where each step replaces declarative components of the query with imperative code [41]. Funke et al. [12] proposed a lightweight intermediate representation (IR) to reduce compilation times for queries by estimating value lifetimes before code generation. The Voodoo IR [35] is a declarative algebra for utilizing many-core architectures and GPUs by generating OpenCL code. Although these approaches are designed for relational DBMSs, query compilation is also applied in several graph DBMSs, like TigerGraph and Neo4j. However, while JIT query compilation is a broad research topic, there is presently no system that utilizes it to hide the memory access latency of PMem.

3 PERSISTENT MEMORY SPECIFIC DESIGN GOALS

This section aims to summarize the observations of several studies as well as our experiments regarding the characteristics and challenges introduced by PMem – in particular, Intel's Optane DCPMMs. Subsequently, we derive general design goals for systems trying to integrate PMem in their hardware landscape. We hope they help others to avoid common pitfalls when conceiving new efficient systems for modern storage hierarchies.

3.1 Characteristics and Challenges

The first three items presented are specific characteristics of Intel's DCPMM technology, while the remaining are explicit challenges that mostly result from PMem and other system peculiarities.

- **(C1) PMem has a higher latency and lower bandwidth than DRAM.** Random access read latency and the read bandwidth of PMem is worse than DRAM by a factor of about three. Persistent writes are also slower than writes to DRAM. PMem bandwidth is about 7 × lower than that of DRAM [42, 48].
- **(C2) Reads and writes on PMem behave asymmetrically.** This concerns several aspects, namely performance, energy consumption, and cell wear. Asymmetrically slower writes cost more energy and lead to wear.
- **(C3) DCPMMs internally work on 256-byte blocks.** They utilize a write combining buffer that is used to reduce

[1] https://www.opencypher.org

[2] http://llvm.org/

write load by trying to combine four cache lines into one 256-byte block write. Interestingly, read operations also benefit when a multiple of the block size is used [42, 48].

(C4) Failure atomicity is only guaranteed for 8-byte aligned writes. The largest failure-atomic store instruction covers only 8 bytes of data, aligned on an 8-byte boundary. Anything larger has to be implemented in software. This means that inconsistencies of data structures due to partial changes in case of system failures and reordering of instructions by the compiler or the CPU have to be avoided.

(C5) PMem allocations are expensive. Compared to DRAM allocations, PMem allocators such as the PMDK allocator need significantly more time [14, 15, 24]. This is mainly due to the necessity of cache line flushes and recovery measures. In conjunction with the higher latencies of PMem, allocations can be –depending on the number of threads– up to 8× slower than on DRAM [24].

(C6) Dereferencing persistent pointers can prevent optimizations. A persistent pointer is a 16-byte structure consisting of a pool identifier (similar to a file path) and an offset in this pool. It was introduced in PMDK and keeps its validity across application restarts. Since this concept of persistent pointers is not integrated into compilers (yet), their handling cannot be automatically optimized as it is the case for volatile pointers [39].

3.2 Design Goals

From the above characteristics and challenges, we can more or less directly formulate corresponding general design goals as follows. Apart from the generic usability of these goals, we will also use them as a foundation for the design decisions in the next section.

(DG1) Algorithmically save writes (C1 & C2). This was one of the first common goals when PMem came up. The idea is to reduce the number of writes by trading them off for more reads. Furthermore, certain intermediate results can be kept in DRAM instead of PMem. In practice, it has been shown that not the number of writes but rather the number of flushed cache lines is decisive.

(DG2) Opt for a DRAM/PMem hybrid storage design (C1 & C2). It has been shown that a pure PMem-only architecture causes too much performance degradation compared to its DRAM counterpart. A hybrid DRAM/PMem approach is therefore highly recommended when seeking the best performance and still requiring persistence [14, 15].

(DG3) Optimize the access granularity to 256 bytes (C3). Besides, the data structures should be aligned to cache lines. Only then a sequential pattern and correspondingly the peak bandwidth can be reached. Everything else can be considered as random access.

(DG4) Prefer failure-atomic writes over logging or shadowing (C4). For this purpose, flushing of cache lines via the `clwb` (cache line write back) instruction and barriers such as `sfence` (store fence) have to be used. However, the number of such barriers should be minimized for best performance. PMDK transactions can be used to simply and universally achieve failure atomicity. However, for performance-critical sections, the underlying logging and snapshotting approach can lead to excessive overhead. Thus, in the long run, an individual realization of failure atomicity with optimally arranged 8-byte stores, `clwb` instructions, and barriers should be preferred.

(DG5) Use group allocations and reuse blocks of memory instead of deallocating (C5). Not every new record in a system should be associated with an allocation. The less frequent allocation of larger blocks or groups can amortize the overhead. Deallocating can also be replaced by suitable free space management. Since they increase the number of allocations, copy-on-write techniques should be replaced by in-place updates or reuse a preallocated space.

(DG6) Avoid dereferencing of persistent pointers (C6). Persistent pointers should preferably only be used during application (re)start for initialization. Afterward, the current valid virtual pointer or application-specific offsets should rather be used. Alternatively, the external location could be converted to a virtual reference once before using it multiple times. In addition, pointer chasing should be avoided as well, as shown in [14, 15].

4 STORAGE MODEL

Essentially, there are two classes of graph data models, namely RDF triple stores and property graphs. RDF stores express everything as triples (subject-predicate-object) which link two nodes or a node to a property value (also called resources and literals). Predicates can therefore be relationships or property keys. Property graphs, on the other hand, consist of explicit node, relationship, and property structures where the properties are directly assigned to a node or relationship. The RDF model creates a lot of redundancies, which could lead to additional write load, which in turn will most likely have a negative impact on PMem performance. Therefore we decided to opt for the property graph model, which is more compact, more expressive, and more efficient to query.

Data Model Definition. In the following, we adopt a property graph model where a graph $G = (N, R)$ consists of nodes N and directed relationships $R \subseteq N \times N$. Each node $n \in N$ is identified by a unique identifier $id : N \rightarrow ID$. Furthermore, a label (used, e.g., as a type descriptor) is assigned to each node and each relationship via a labeling function $l : \{N \cup R\} \rightarrow L$ where L is the set of labels.

Properties are represented as key-value pairs $(k, v) \in P$ with $P = K \times D$ where K denotes the set of property keys and D the set of possible values including numbers, strings, etc. To each node and relationship, a set of properties can be associated via $p : \{N \cup R\} \rightarrow \mathcal{P}(P)$ where $\mathcal{P}(P)$ denotes the power set of P.

4.1 Design Decisions

The above data model is implemented by storing the graph in node, relationship and property tables maintained in persistent memory. For efficient data access, the specific characteristics of current PMem technology as mentioned in Sections 1 and 3 have to be taken into account. The application of our derived design goals led to the following key design decisions:

(DD1) Each of the tables is managed as a **linked list of chunks** where a chunk is a fixed-sized array (cache-line aligned and a multiple of 256 bytes) of records. To reuse the

space of deleted records, standard free space management using a persistent list is implemented. This way, tables can dynamically and efficiently grow or shrink for updates, by allocating/deallocating chunks (DG3, DG5).

(DD2) A chunk stores **equally-sized records** of the same type (nodes, relationships, properties). Thus, records can be **addressed via their offsets**. Similar to a sparse index, an additional persistent lookup table allows efficient access to chunks based on the record offset (DG1, DG6). Note that we use array offsets because they can be represented as 8-byte integers instead of 16-byte persistent pointers. This not only saves space but also allows for **failure-atomic stores** and avoids costly dereferencing (DG1, DG3, DG6).

(DD3) In order to represent nodes and relationships as equally-sized records, **properties are outsourced** to a separate table. Furthermore, all variable-length values (e.g., strings) are **dictionary encoded**. Both lead to a reduced number of write operations (DG1).

(DD4) The **connections** between nodes and their relationships as well as their properties are represented via array **offsets** instead of (persistent) pointers. Because relationships are directed, each node refers to its list of both outgoing and incoming relationships, also via offsets.

(DD5) The **storage model** is designed **hybrid** both for secondary indexes and for transaction management (DG2). Further details are provided below.

4.2 Key Data Structures

In the following, we give an overview of the key data structures to represent the property graph model and further structures necessary to realize our design decisions and achieve a great performance.

Nodes, Relationships, and Properties. Fig. 1 illustrates the primary storage structures of a persistent graph which we have implemented using Intel's PMDK [17]. The highlighted row illustrates a respective node or relationship record. On top of both the node and relationship table, an additional sparse index is used which maps the identifiers of the first record of each chunk to their corresponding memory location. For each chunk there is a bitmap to indicate free and occupied record slots, enabling an efficient reclamation of deleted entries. The chunks are linked by a persistent pointer to allow the scanning of all data. Node records consist of a label, the offset of the first incoming and first outgoing relationship, as well as the offset to their properties. Relationship records also have a label as well as the offset to their properties. Furthermore, they store the location of the source and destination nodes that they connect. Optionally, relationship records hold offsets to the next relationships of their source and destination node. Note that the records for nodes and relationships contain a few additional fields needed for transaction processing which are described in Section 5. In total, this results in a record size for nodes and relationships of 56 and 72 bytes respectively.

The properties are stored in a separate chunked table as key-value pairs. These are grouped in batches, each belonging to a single node or relationship, to obtain cache-line-sized records. In order to allow variable-length key-value pairs, string types are stored as dictionary codes. If there are more properties for a single node or relationship, the property record links to the next entry. These data structures resemble the typical storage layout

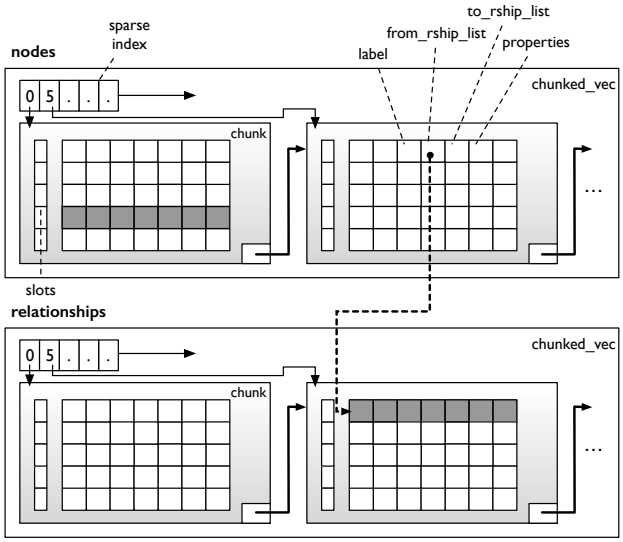


Figure 1: Graph data structures

of disk-based, table-oriented systems such as SQL databases or even graph databases like Neo4j. However, in our case, table chunks are not copied between disk and memory but instead accessed directly in PMem. In addition, nodes, relationships, and associated properties can be addressed individually via their identifiers/offsets.

Dictionary. As mentioned before, to allow for variable-length labels, property keys, and values, a dictionary is used. This compresses strings and, thus, reduces space and write overhead as well as ensures that records remain addressable by offset. Furthermore, the comparison of codes instead of strings speeds up operators such as filters. The dictionary consists of two hash tables for bi-directional translation to make lookups fast. These must be kept persistent, in case of failure, since the codes and strings are not stored elsewhere. An alternative could be to only store one of the hash tables in PMem and rebuild the other DRAM-resident part. Depending on the workload (either more inserts or more queries), the more frequently used table should be kept in DRAM.

Hybrid Indexes. The table-based storage model is useful for lookups on physical node/relationship identifiers (which represent array offsets) as well as scan-intensive processing where large parts of the nodes or relationships are visited. However, for lookup queries on node/relationship properties, scans are too expensive. In order to accelerate these queries, we additionally provide B^+-Tree indexes. An index can be constructed on nodes with a given label and for a property. The values of these properties are used as keys in the index. Since the indexes are secondary data structures that can be rebuilt in the event of a failure, they do not have to be completely persistent. To still have a good compromise between recovery and query performance, we opted for a DRAM/PMem hybrid approach (selective persistence) similar to [18, 31, 47]. In particular, this means that the leaf nodes are stored in PMem and the inner nodes in DRAM, resulting in a maximum of one PMem-resident node being read per lookup (if not already cached by the CPU) and significantly reduced recovery time. This has an additional economic advantage since less DRAM is used, which we expect to be more expensive than

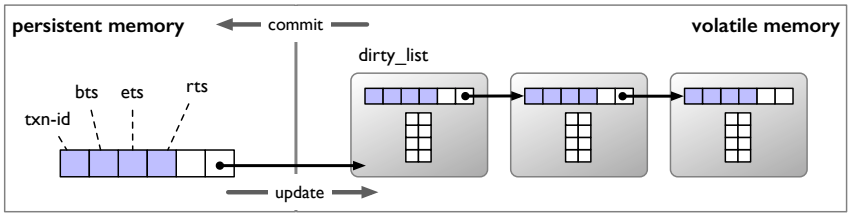


Figure 2: Structure of transactional data

PMem in the near future. In accordance with DG3, all nodes on PMem are cache-line-aligned and a multiple of 256 bytes. For analytical queries, multi-dimensional index structures optimized for PMem could also be used where properties represent the dimensions [18].

5 TRANSACTION PROCESSING

An HTAP architecture requires high-performance concurrency control mechanisms. Several studies in the past [34, 46] have shown that DBMSs with multi-version concurrency control (MVCC) exhibit higher concurrency than their single-version counterparts. Here, transactions can be concurrently executed on different versions of the same object, thus increasing the overall transaction throughput especially when the transactions are long-running and contention is high [22]. This also allows for scalability and efficient utilization of modern multi-core CPUs. MVCC is implemented differently by different DBMSs, each making certain design decisions in order to optimize for its target workloads. The interplay between these design decisions ultimately results in computation and storage overhead trade-offs. Below, we discuss how we implemented these MVCC design decisions in a PMem setting to achieve our design goals presented earlier in Section 3.

5.1 Concurrency Control Protocol

Existing concurrency control (CC) protocols such as two-phase locking (2PL), optimistic concurrency control (OCC), or timestamp ordering (TO) can essentially be used in a multi-version setting. We chose MVTO as our CC protocol. With our MVTO implementation, we support updates of an arbitrary number of objects within a single transaction and achieve snapshot isolation guarantees. Note, that we use MVTO here mainly as an example to evaluate how an MVCC protocol implementation can exploit and address the specifics of PMem. However, in principle, the main concepts should apply to implementations of other protocols too.

There is a transaction identifier (timestamp), `txn-id`, given to each transaction at the beginning of the transaction that uniquely identifies it. Each data object maintains meta-data fields for concurrency control purposes. To this end, we extend the data structures of nodes and relationships, as shown in Fig. 2, by additional *persistent* fields – `txn-id`, begin timestamp `bts`, end timestamp `ets` and read timestamp `rts` – and a volatile field – pointer. The `txn-id`-field is used for write-locking, by way of coordinating which versions are valid for which write-transactions. By default, it is set to zero except if the object is locked by a write-transaction, where it is set to the transaction's `txn-id` using a CaS instruction [22]. The begin timestamp and end timestamp fields mark the validity of an object for access by a read-transaction, while the read timestamp indicates the latest transaction that read it. The pointer field stores a *volatile* pointer to a list of dirty objects (i.e., in DRAM) to address (DG1) and (DG2). Alternatively, the `bts`, `ets`, and `rts` fields and perhaps also the `txn-id` of the current version could be moved to DRAM in order to reduce the persistent record size. These fields could then be re-initialized during recovery (or during the first access after a failure). However, this could also be disadvantageous because the transaction information of the current version would always have to be retrieved with another random read in DRAM.

Write transaction. A transaction T always updates the latest version of an object o. It creates a new version o_{i+1} of the object if no other transaction has a lock on o_i and o_i has not been read by a more recent transaction (i.e., the transaction identifier $id(T) > \texttt{rts}(o_i)$). Otherwise, T aborts. The `txn-id` field of o_{i+1} is set to $id(T)$. In case of an update, o_{i+1} is kept in the dirty list in volatile memory until commit. If the transaction inserts a new object, this object is already stored in the persistent array (i.e., in PMem), but still locked until the end of the transaction.

Read transaction. A transaction T reads version o_i of an object for which $id(T)$ is between the `bts` and `ets`, i.e., $\texttt{bts}(o_i) \leq id(T) < \texttt{ets}(o_i)$, and which is not locked by another active transaction. Thus, the object is accessed in PMem first (representing the most recent committed version) and if this is not the version valid for T then the dirty list in volatile memory is traversed to retrieve the correct version. In case of a lock held by another transaction, the transaction is aborted. Upon reading o_i, the `rts` field is updated to $id(T)$ unless $\texttt{rts}(o_i) \geq id(T)$. In this case, the transaction reads an older version without updating `rts`.

Commit. For commit of a transaction T, the timestamp fields of the updated object version o_{i+1} are set accordingly: `bts` to $id(T)$ and `ets` to `INF` and for the previous version o_i, the field `ets` is set to $id(T)$. In the case of delete, `ets` of the deleted version o_{i+1} is set to $id(T)$ instead. If the object was newly created, however, it is simply unlocked (i.e., resetting `txn-id` to 0). Otherwise, o_{i+1} has to be copied back to PMem. In order to guarantee failure atomicity, this memory copy has to be performed atomically. This can be implemented in different ways. One approach is to rely on the solution provided by the Intel PMDK to atomically update and persist data that is larger than the power-fail atomic size or portions of data that are non-contiguous. PMDK uses transactional operations for memory allocation, freeing, and setting. Internally, these transactions are implemented via redo logging to ensure the atomicity of memory allocations and undo logging for transactional snapshots [40]. Other approaches are, e.g., using Multi-Word CaS instructions such as PMwCAS [44] which allows atomically changing multiple 8-byte words on PMem. In our current implementation, we use the PMDK solution for the sake of simplicity (DG4). However, this comes with a small overhead.

5.2 Version Storage

A transaction updating an object version o_i creates a new version o_{i+1} by making a copy of o_i and appending it to the front of the list of dirty versions (i.e., *version chain*). It then performs all updates on o_{i+1} in DRAM until commit. Keeping all uncommitted data in volatile memory is a design decision we made in order to minimize the number of writes to PMem (DG1, DG2). This hybrid DRAM/PMem approach allows for the creation of all versions by transactions to be a volatile copy instead of the more expensive copy to PMem, and also allows for all the write operations that occur during the lifetime of a transaction to be performed at DRAM latency until the transaction is to commit when the updates are finally persisted in PMem. Note that a dirty object has the same structure as its committed version but with a different validity interval (as specified by the range [bts, ets]).

5.3 Garbage Collection

In our current implementation, we use Transaction-level Garbage Collection (GC), where storage space occupied by dirty versions that are not going to be used anymore is reclaimed at transaction-level granularity [46]. A node or a relationship maintains a volatile dirty list, only if there is a valid dirty version of it. A dirty version is not used anymore if it becomes invalid (i.e., the transaction that created it aborts) or if it is no longer *visible* to any active transaction (i.e., its ets $<$ $id(T)$ of the oldest active transaction T). All empty or unused dirty lists are discarded during a commit. If the storage space to be reclaimed is in PMem, either because a committed transaction deleted the object or the object was inserted by an aborted transaction, we do not deallocate the record slot(s). Rather, we simply mark it with a bitmap as free for later reuse (DG5).

6 QUERY PROCESSING

The characteristics of PMem have several implications for query processing. First of all, data access is no longer block-oriented and, therefore, has to be optimized for sequential access. The direct and byte-addressable access is very similar to in-memory databases. In graph databases, this is particularly useful for traversal operators. However, as mentioned above, reading from PMem is slower than from DRAM. Hiding this higher latency requires efficient cache utilization, multithreaded processing, and various execution modes.

6.1 Push-based Approach

We address these requirements by a multithreaded push-based query engine. Our engine provides a set of graph-specific algebra operators [16] such as NodeScan, RelationshipScan, and ForeachRelationship; as well as standard relational operators like Filter, Project, and several Join variants. As every operator is implemented and ahead-of-time (AOT)-compiled, i.e., available at run-time, the engine is able to interpret queries (given as graph algebra expressions) directly by calling these operators with the required parameters. Processing a typical traversal query (Match in Cypher) is initiated by scans on the node or relationship tables including filters. For each node satisfying the optional filter condition the traversal operation is applied, i.e., the NodeScan operator forwards the current node to the next operator ForeachRelationship, and so on. (Fig. 3).

Though traversals could be also implemented using joins of a standard relational query engine [11], the ForeachRelationship leverages the direct addressability of data in PMem. As described

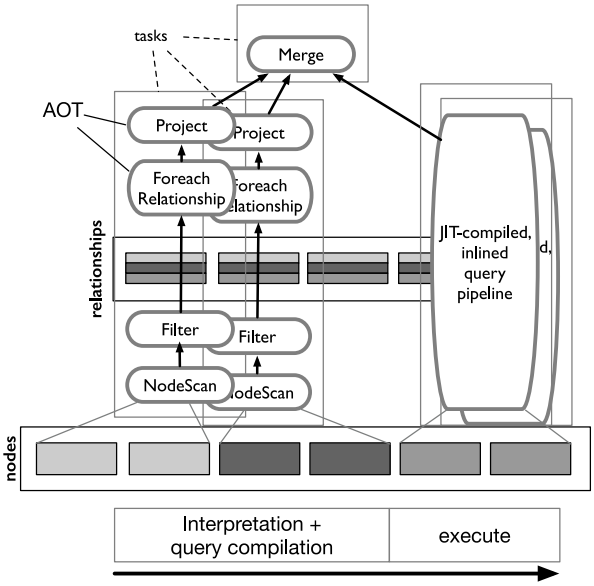


Figure 3: Query execution plan

in Sect. 4.2, node records contain the persistent addresses (offsets) of their relationships, which in turn store the address of the sibling nodes, and are used to traverse the path. This avoids the problem of join escalation during traversals.

For query parallelization, we leverage a task model. Scans are performed as parallel scans: Each task processes a range of the node/relationship tables. Thus, all subsequent operators following the scan are also performed within this task until a pipeline breaker like a join or sort operator is reached. This way, we follow the morsel-driven parallelism approach [23].

For using indexes in query processing, an appropriate IndexScan operator is provided that performs a lookup or range scan on the B$^+$-Tree, extracts the matching nodes from the node tables, and passes them to the subsequent operator pipeline.

6.2 Just-In-Time Query Compilation

Besides the AOT compiled query engine, we implement a JIT query compiler that transforms graph algebra expression into machine code. Traditional query execution engines use a query interpreter to execute query statements. This has several drawbacks that reduce the resulting performance. An interpreter relies on AOT compiled code, which means that the appropriate methods must be available for every possible occurrence of a particular tuple element type. Furthermore, a query interpreter is not able to recognize equal or redundant instructions. Particularly for operators that lead to variable cardinalities, like selections or aggregations, it introduces additional overhead. Query compilation is an approach to tackle these issues. Our approach aims to compile given graph algebra expressions into highly efficient machine code using the JIT compilation technique [20]. As the compiling framework, we chose the LLVM compiler infrastructure because it provides numerous relevant features for the JIT compilation, reliable performance, and portability to several architectures. Moreover, the LLVM IR provides an instruction set suitable for the implementation of all the abstractions needed for our graph query engine. One significant requirement for the JIT query compiler is the fulfillment and compliance with the formulated design goals (DG1-DG6). This is mainly done by reusing

(calling) AOT-compiled code, e.g., access methods to nodes or methods for transaction processing. Thereby, the code generation effort will be reduced because it is already compliant with the design goals and optimized by the AOT compiler.

Similar to the approach presented by [28], we aim to process intermediate tuple results as long as possible in the CPU registers. In order to achieve this, it is necessary to transform the complete query pipeline into a single LLVM IR function. Here it becomes apparent that a transformation from graph algebra to machine code can be easily accomplished with LLVM IR. However, to ensure reliable performance, we identified the following requirements for the IR code generation that must be met.

(1) Minimize stack allocation and avoid heap allocation.
(2) Process initializations only at the first entry point of the IR function.
(3) Process type information at (JIT) compile-time.
(4) Provide full compatibility to the AOT execution engine.

One significant advantage of query compilation over interpretation is that the tuple element type information can be handled at compile-time. The consequence of this is the absence of type conversions at run-time as code can be generated for individual types.

Starting from a graph algebra expression that forms an operator tree, each operator will be transformed into LLVM IR code. Further, each operator provides at least an *entry* and a *consume* IR basic block, representing the operator's start and end points. Complex operators comprise more basic blocks for the actual tuple processing, e.g., Join. Though, the general control flow starts at the entry basic block. After processing in further basic blocks, the control flow branches to the consume basic block to push the results to the next operator. A branching instruction links each consume basic block with the entry basic block of the succeeding operator, forming an inlined query pipeline. Fig. 4 illustrates the transformation process, starting from a query plan in the form of graph algebra. Furthermore, it shows each operator's return path, which is, for most cases, the loop header of the previous operator. The finish operator will be called after the complete scan. Depending on the query, it invokes the function return or the next query pipeline.

The query engine's current implementation provides two access paths for the query pipeline: the NodeScan and Create operator. Code generation for these operators is basically the same as for the normal operators. Both contain at least the entry and consume basic block. As an access path is always the first operator in the pipeline, it must also provide the actual generated

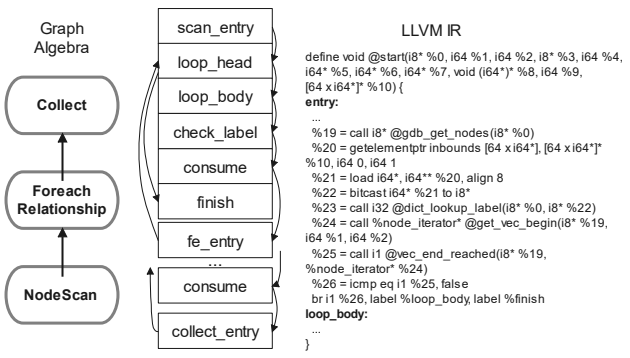


Figure 4: Graph algebra to LLVM IR transformation

function's entry point. Due to the reason that the code generator transforms the complete pipeline into a single IR function, memory allocation must be carefully handled. For this reason, the access path initializes all relevant values for the complete query pipeline, e.g., number of nodes, projection keys, or global constants.

The code generation for joins requires additional work. A join operator comprises two inputs. For now, we consider the right sub-pipeline of the join as the side which will be materialized. Consequently, it requires the prior execution of the right sub-pipeline. However, the actual code generation starts from the left sub-pipeline in order to minimize tree traverses. Whenever the code for an access path is generated, it checks if the current function is already initialized. If this holds, it swaps the function entry basic block with its own and connects the finish basic block of the second access path with the entry basic block of the previous access path. This enables the handling and execution of multiple query pipelines within a single function.

IR Code Generation. We use the *visitor design pattern* to generate the appropriate IR code for each operator. Further, this enables the extension of the query engine in future work. Each operator derives from a base class and implements a `codegen` method for code generation. The query engine calls the *visitor* to start the code generation process, which calls all the operator's `codegen` methods recursively. We implement several IR abstractions that help to generate IR code with more ease. Due to the reason that most operators rely on loops, we provide two IR loop abstractions. The `while_loop` abstraction is used for the iteration through a chunked vector. It receives the vector, the current iterator, the succeeding basic block, and the actual loop body as function arguments. The other loop abstraction is the `while_loop_condition`, used for the iteration as long as a condition is valid. Further, our abstraction set contains methods that generate code to extract label codes or property values. The operator IR code is based on these abstractions and mainly mimicking the push-based processing described previously. An additional structure is built to provide the type of tuple element at code generation and the appropriate register value. The code of the next operators is generated according to the type of the previous tuple element. For example, the projection operator uses the proper node functions if the last result tuple element is a node. This handling allows for generating code without much effort at run-time.

JIT Compilation. We extended the JIT compiler of LLVM to further features. First of all, our JIT query engine can persist already compiled code to PMem. This has the advantage that no further compilation is required for subsequent runs of a query. For that purpose, a persistent and concurrent hash map is used. The compilation output of the JIT is a binary object file that will be linked with the current database instance. Usually, this file is located in a memory buffer in the volatile memory. Before the compilation process, the query engine generates a unique query identifier that comprises the operators' identifiers, which will be used to lookup the persistent hash map for already compiled code. If the code is found, it will be linked with the current database instance. Otherwise, the compilation process of the query starts. The compiled code will be persisted in PMem after its compilation, using the query identifier as a key for the hash map.

A major advantage of JIT compilation is the ability to optimize the IR code at run-time. The LLVM framework provides a convenient approach for IR code optimization. Several LLVM

optimization passes can be used for this purpose. An analysis of the IR code reveals that it comprises mainly of loops and pointer arithmetics. Therefore, our optimization strategy focuses on these constructs. The following optimization pass cascade is used to further optimize the code at run-time:

- *Promote Memory To Register* transforms instructions that allocate stack memory into register values. This makes the IR code generation convenient and compliant with the requirement (1).
- *Control Flow Graph Simplification* merges and deletes basic blocks if they have common or no predecessors.
- *Loop Unrolling* removes the overhead of loops by explicit extraction of the body to multiple instructions.
- *Dead Code Elimination* eliminates unreachable code.
- *Instruction Combining* combines redundant instruction to form smaller and faster code with the same effect.

Additionally, the IR code is optimized with the standard C++ aggressive optimization (-O3).

Adaptive Execution. While the compiled query code itself is fast, the compilation time should also be considered. Notably, when executing short-running queries where only a small portion of data is touched, the compilation time will be longer than the actual execution time. In order to hide the compilation time as well as memory access latency of PMem, we additionally support an adaptive query processing approach, which is illustrated in Fig. 3. In contrast to the approach by [21], the adaptive execution can switch between only two modes, which is currently sufficient for our engine. The interpretation mode is always initiated first at query execution. This mode uses AOT-compiled database code to execute the query. Similarly, the visitor design pattern is used to transform the given algebra query plan into the interpret functions. These functions are then linked together, forming a cascade of functions that execute the actual query. The downside of such an approach is the additional (AOT-compiled) code overhead because every operator and its varieties must be available at compile-time. During adaptive execution, the query engine switches to the JIT mode after compilation.

We take advantage of the morsel-driven parallelism for the actual switching procedure, where morsels are pinned to a single task and pushed into a task pool. The working threads pull a task from the pool and execute the task function (the query) on the pinned morsel.

We implement the task function as a static function. As the execution always starts in the interpretation mode, it will be initialized to the appropriate function, which invokes the interpretation. While the query is executed in the interpretation mode, a background thread compiles the query plan into machine code. The compilation process emits a function that processes the query plan into machine code. As soon as the compilation is done, it redirects the static task function to the compiled function. The next pulled task from the pool will execute the compiled query function.

7 EVALUATION

In this section, we report the results of a set of experimental evaluations whose research goal is threefold:

1. We evaluate our PMem-based HTAP engine and show the effectiveness of our design decisions to exploit PMem characteristics for graph processing. In this context, we aim to investigate, on the one hand, the benefits of using persistent memory for graph processing. On the other hand, we compare our system to disk-based and DRAM-based solutions (§ 7.3).
2. We compare the speed of volatile, persistent, and DRAM/PMem hybrid B^+-Tree index lookups. We quantify the recovery overhead of our hybrid index (which we expect to be insignificant) as a trade-off for increased query performance (§ 7.4).
3. We evaluate our JIT query compilation approach. We demonstrate when and how much it enhances the performance of transactional queries. We expect the JIT compilation to yield benefits especially for long-running and more complex queries (§ 7.5).

7.1 Environment

For the experiments, we used a dual-socket Intel Xeon Gold 5215 server with 10 cores each at max. 3.4 GHz. The server is equipped with 384 GB DDR4 RAM, 1.5 TB Intel Optane DCPMM, and 4 × 1.0 TB Intel SSD DC P4501 Series connected via PCIe 3.1 x4. The server runs CentOS 7.8 (Linux 5.7.7 kernel). The operating mode of the PMem modules is set to AppDirect which allows us direct access to the devices. On the PMem DIMMs, we have created an ext4 file system and mounted it with the DAX option to enable direct loads and stores bypassing the OS cache. For accessing PMem, we used the Intel PMDK version 1.9.1 and libpmemobj-cpp[3] version 1.11. The JIT compilation was done with LLVM version 11.

7.2 Workload & Setup

The Linked Data Benchmark Council (LDBC) specifies benchmarks and benchmarking procedures and also verifies and publishes benchmark results [10]. The LDBC-Social Network Benchmark (SNB) models a social network comprising of different entity types interconnected by relationships – both with property types and property values. Activities of persons are represented as messages about topics or tags that are posted in forums moderated by unique persons. Persons like messages, have interests in tags, are members of forums, and make comments in response to posts or other comments. Message activities are the bulk of the data on the social network. There also are places and organizations to which a person is connected via residence, study, and work relationships. The LDBC-SNB defines an Interactive Workload and a Business Intelligence Workload. The Interactive Workload comprises of three classes of queries: (1) Interactive Complex Read Queries that are relatively complex and traverse a fair portion of the graph data, (2) Interactive Short Read Queries that perform lookups and short traversals within the neighborhood of a node, and (3) Transactional Interactive Update Queries that perform transactional insertions and updates of node and relationship objects [10].

We generated and used the LDBC-SNB data [2] at scale factor (SF) 10 as our benchmark data. As the focus of this paper is on transactional graph processing, not graph analytics, we selected the LDBC-SNB Interactive Short Read (SR) and the Interactive Update (IU) query sets as query workload for our experiments.

[3] https://github.com/pmem/libpmemobj-cpp

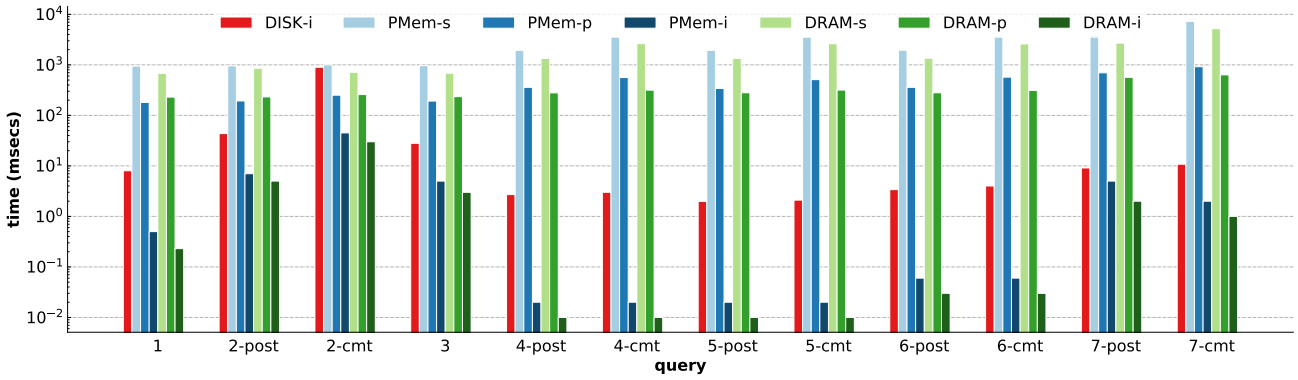


Figure 5: Results for SNB Short Reads

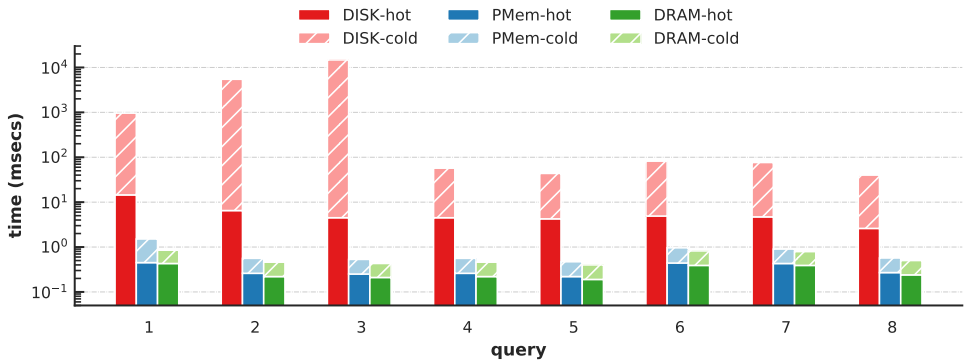


Figure 6: Results for SNB Interactive Updates

7.3 Benefit of PMem

We first want to evaluate how much the design decisions in our PMem-optimized graph engine and our implementation of transaction processing reduce the overhead of PMem in our system (denoted as PMem in the figures) compared to a pure DRAM-based in-memory implementation of it. Moreover, we want to compare the performance gains brought about by the lower access latency of PMem compared to a DISK-based system, in addition to providing persistence. To this end, we employ two baselines: A disk baseline (represented as DISK), which is an open-source native graph database where we stored all the primary data on SSD and created an additional DRAM index. For the DRAM baseline (depicted as DRAM), we adapted our system to additionally run in a pure volatile mode where we keep data entirely in DRAM. We expect our system to outperform the disk-based system. With regards to the DRAM baseline, we anticipate to bridge the performance gap with our PMem-conscious design and achieve a near-DRAM performance while providing persistence, especially for hot runs.

Interactive Short Reads. Fig. 5 shows the query execution times for the SR query set. The execution times are average times of 50 runs on hot data, each with a different input ID parameter. post and cmt (short for *post* and *comment* respectively) represent the two subclasses of a message entity. For PMem, we show the execution times without indexes for single-threaded execution (PMem-s), multi-threaded execution (PMem-p) as well as with indexing support (PMem-i). We employ similar denotations for our DRAM baseline: DRAM-s, DRAM-p, and DRAM-i. For the disk baseline, we also conducted executions with index support and report the performance numbers for hot runs (i.e., when the data is in DRAM), denoted by DISK-i. We used our hybrid DRAM/PMem implementation of the B$^+$-Tree (Section 4) for PMem, while for DRAM, we used a volatile B$^+$-Tree. We maintain the same set of indexes throughout our experiments.

The results in Fig. 5 show that exploiting PMem-specific characteristics in storage architecture and transaction processing can significantly bridge the performance gap between DRAM and PMem. It can be noted that for multi-threaded execution of some of the queries, the execution times are very close since the SR queries are short-running and the PMem latency is already hidden by the CPU caches for hot runs. An interesting research direction is thus to investigate this in the context of graph analytics, where queries are compute-intensive, long-running, and navigate across a significant portion of the graph. While the results show performance improvements of multi-threading both for DRAM and PMem, however, indexes have a stronger influence. Unlike graph analytics that significantly benefit from parallel execution, interactive queries like SR and IU benefit more from indexes, as they are essentially lookup queries whose execution time overhead comes mainly from scanning the tables of record chunks to retrieve the start node object. As a result, we compare the performance of indexed query execution both on our system and on the DISK baseline. We can see from the figure that our PMem-based system outperforms the disk-based system for indexed execution in all the queries, as we had expected.

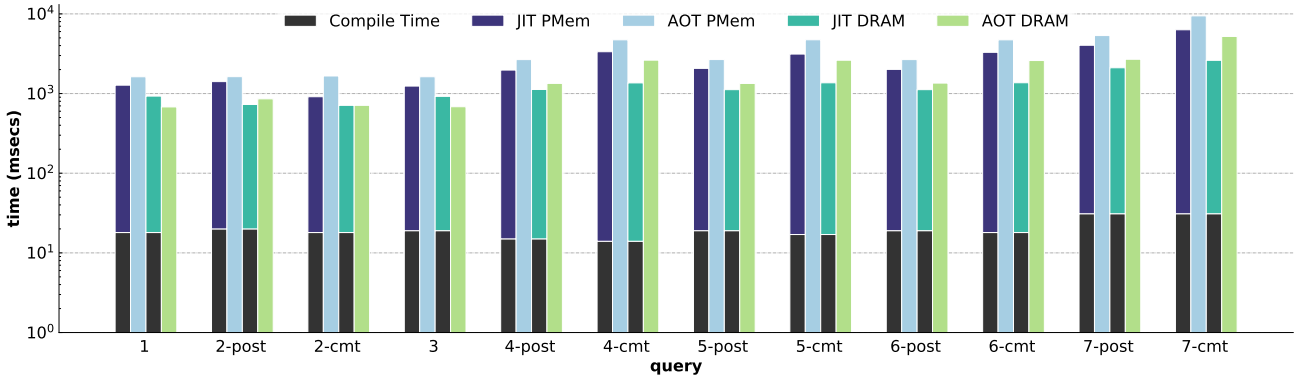


Figure 7: Results for SNB Short Reads with Single Threaded Execution

Interactive Updates. We maintain the indexed query execution and present the execution times for the IU query set with indexed support in Fig. 6. Here, we measured both the times to execute the update queries as well as times for the transactions to commit (i.e., notably, persisting the updates in PMem). Similar to the SR queries, we took the average execution time of 50 runs on hot data with varying object property values as input parameters. In addition to results on hot data, we also present the execution times for cold runs, i.e., for the first query runs. The results show our PMem-based system not only outperforms the disk-based system by an order of magnitude even for hot runs but also performs insert and update operations at near-DRAM latency. For hot data, it is even closer.

Overall, the results of Fig. 5 and Fig. 6 show that in direct comparison with the DRAM variant, our hybrid approach of MVTO implementation to address the specifics of PMem adds only a marginal overhead. This validates our MVCC design decisions of Section 5 and also obviates the need for showing the results of a pure PMem implementation which has an overhead of maintaining dirty versions on PMem.

7.4 Indexes and Recovery

We evaluated index performance and recovery by way of comparing our hybrid index that keeps inner nodes in DRAM, trading-off recovery for improved performance, against two baselines. One a volatile index that keeps all nodes in DRAM and the other a persistent index that stores all nodes on PMem. We evaluated them based on the average time for indexed scans in the SNB SR queries. To study the performance differences, we measured the time to lookup and retrieve a node ID from the appropriate index. Fig. 8 shows the average lookup time for the persistent, volatile, and hybrid indexes - denoted respectively as PMem, DRAM, and Hybrid. The lookup times are averages of all ID value lookups of nodes with the same label type (Person) in all the respective SR queries. The hybrid approach enhances the lookup performance by 2x while keeping the recovery time as low as 8 ms, in comparison to the complete volatile index build time of 671 ms. This recovery overhead would also be necessary for each index created on specific properties. Added up, the overhead of completely rebuilding the indexes in the volatile case is comparatively drastic. Therefore, we see the hybrid variant as a good compromise between runtime performance and recovery.

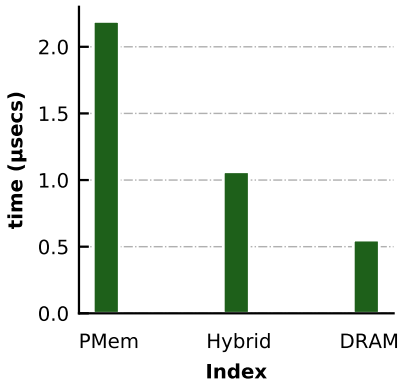


Figure 8: Average Time per Lookup of Persistent, Hybrid, and Volatile Indexes

7.5 JIT

The final part of our evaluation focuses on the JIT query compilation approach. The first two benchmarks show the capability of JIT-compiled code itself, without any mechanism to hide the compilation time. For this purpose, we execute the interactive read and update queries from the SNB. Thereafter, we examine the gain from adaptive execution. Although we expect it to be much more efficient for analytical and long-running queries, it is insightful to see the benefits on short and transactional queries in comparison to AOT-compiled code. In particular, the combination with PMem could make this approach profitable even for short-running queries.

Interactive Short Reads. Fig. 7 shows the results for the SR executed with the JIT query engine. We calculated the average execution time of 50 runs on hot data with different parameters. The queries are executed single-threaded without indexes. The compilation time of the queries is only a few milliseconds. As the number of operators increases, the compilation time increases by only a few milliseconds. However, the results show clearly that the JIT-compiled is always faster than the AOT-compiled code. The JIT-compiled code is mostly even faster when the actual compilation-time of the query is included. Especially more complex queries, like 7-POST and 7-CMT, can benefit from the JIT compilation approach.

Interactive Updates. The results for the IU executed with the JIT query engine are shown in Fig. 9. There are not many optimization possibilities for the generated IR code, as the queries are

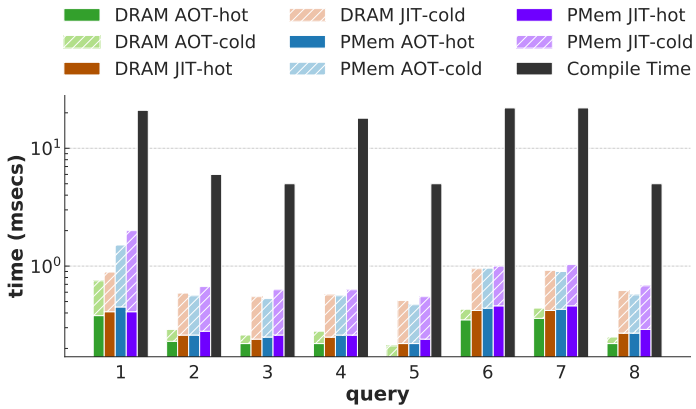


Figure 9: Results for SNB Interactive Updates executed with JIT compiled code

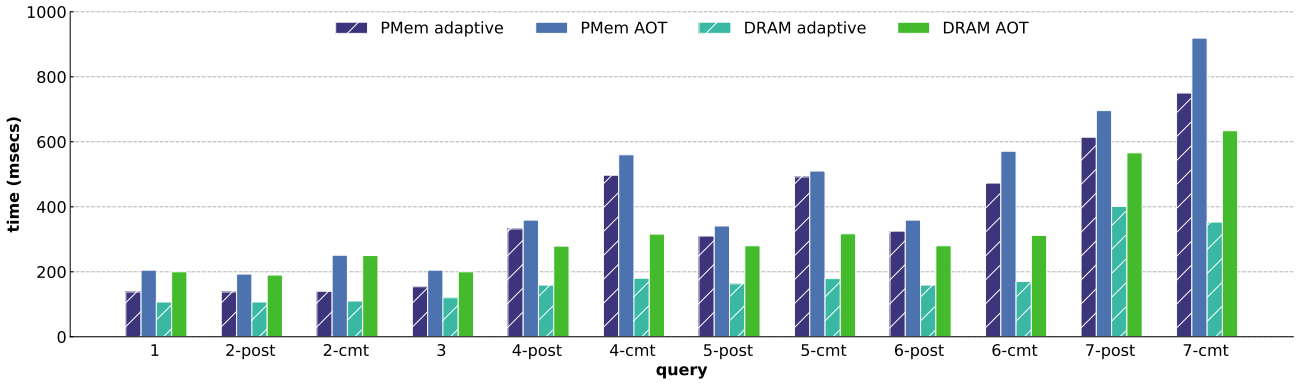


Figure 10: Results for SNB Short Reads with Adaptive Execution

short when index support is enabled. Executing these queries using scans and selections shows similar behavior to the benchmark before. However, here we focus on code for short queries, where the execution time is less than the compilation time. JIT code executed on cold data is noticeably slower, while the resulting performance on hot data is similar to the AOT code. However, executing these queries with the JIT compilation approach shows that it is not always the best option to generate code during runtime for executing a query. The compilation time for these short queries is much higher than the actual execution time. Furthermore, executing these short queries with the adaptive approach leads to the execution of the query pipeline using the AOT-compiled code entirely, which corresponds to the results of the AOT code in Fig. 9.

Adaptive Query Executions. The previous benchmarks show the capability of executing JIT-compiled queries. It is clearly visible that the JIT-compiled code itself outperforms the AOT code on DRAM and PMem. However, waiting for the compilation of the query limits the performance improvement of this approach. The adaptive query execution approach eliminates this problem by executing the AOT code while query compilation is done in the background. Additionally, this is useful to hide the memory access latency of PMem. The next benchmark compares the adaptive query execution approach using morsel-driven parallelism with multi-threaded AOT-compiled query execution. Similar to the previous benchmarks, we execute each query on DRAM and PMem. The results in Fig. 10 show that the adaptive execution is always faster than the multi-threaded AOT execution. The execution on PMem can particularly benefit from the adaptive approach. The additional latency introduced by PMem leads to an earlier execution of the fast JIT-compiled code in the query pipeline stage, which enhances the query processing. For the queries 1, 2-POST, 2-CMT, and 3, it leads to similar execution times for DRAM and PMem. The adaptive approach provides faster query execution times for most queries and in worst-case similar performance than multi-threaded AOT code. More complex queries can benefit even more from the adaptive approach as there is more space for code optimization, like for the queries 7-POST and 7-CMT.

8 CONCLUSION

Persistent memory represents a promising technology for data management solutions whose efficient use requires rethinking data structures and architectures. In this work, we have presented the first results of our PMem-based graph engine for hybrid transactional/analytical workloads. Based on the characteristics of PMem technology, we have discussed, implemented, and evaluated design choices regarding storage structure, transaction, and query processing. The promising results using the LDBC-SNB interactive short read and update query sets show that a PMem-based storage engine that is well-optimized for PMem characteristics incurs only a marginal performance overhead compared to a pure in-memory solution. The main benefits are, among others, the competitive performance without the need to

keep large parts of the data in (volatile) main memory (resulting in constant answer times both for cold and hot data) as well as near-instant recovery guarantees. Additionally, the results have shown that in comparison to AOT-compiled query execution, JIT compilation speeds up query processing when the compilation time is less than the execution time. Particularly, adaptive compilation further enhances query execution performance by hiding PMem access latency. In our ongoing work, we plan to investigate the behavior of complex graph analytics and highly concurrent updates. Moreover, there are several opportunities for further performance improvements, e.g., by employing more hybrid DRAM/PMem approaches such as for dictionaries.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG) in the context of the project "Hybrid Transactional/Analytical Graph Processing in Modern Memory Hierarchies (#TAG)" (SA 782/28-2) as part of the priority program "Scalable Data Management for Future Hardware" (SPP 2037) and by the Carl-Zeiss-Stiftung under the project "Memristive Materials for Neuromorphic Electronics (MemWerk)".

REFERENCES

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *SIGMOD*. 431–446.

[2] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020).

[3] Renzo Angles and Claudio Gutiérrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1 (2008), 1:1–1:39.

[4] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB* 11, 5 (2018), 553–565.

[5] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *PVLDB* 10, 4 (2016), 337–348.

[6] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *CoRR* abs/1910.09017 (2019).

[7] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an efficient RDF store over a relational database. In *SIGMOD*. 121–132.

[8] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB* 8, 7 (2015), 786–797.

[9] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. 2020. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *PVLDB* 13, 9 (2020), 1598–1613.

[10] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*. 619–630.

[11] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *CIDR*.

[12] Henning Funke, Jan Mühlig, and Jens Teubner. 2020. Efficient generation of machine code for query compilers. In *DaMoN @ SIGMOD*. 6:1–6:7.

[13] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *PVLDB* 13, 8 (2020), 1304–1318.

[14] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. 2020. Data Structure Primitives on Persistent Memory: An Evaluation. *CoRR* abs/2001.02172 (2020).

[15] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. 2020. Data structure primitives on persistent memory: an evaluation. In *DaMoN @ SIGMOD*. 15:1–15:3.

[16] Jürgen Hölsch and Michael Grossniklaus. 2016. An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In *EDBT/ICDT*.

[17] Intel Corporation. 2020. Persistent Memory Development Kit. http://pmem.io/pmdk. Online, accessed December 2020.

[18] Muhammad Attahir Jibril, Philipp Götze, David Broneske, and Kai-Uwe Sattler. 2020. Selective Caching: A Persistent Memory Approach for Multi-Dimensional Index Structures. In *HardBD & Active @ ICDE*. 115–120.

[19] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *USENIX ATC*. 993–1005.

[20] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *PVLDB* 11, 13 (2018), 2209–2222.

[21] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *ICDE*. 197–208.

[22] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5, 4 (2011), 298–309.

[23] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*. 743–754.

[24] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *PVLDB* 13, 4 (2019), 574–587.

[25] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling Low Tail Latency on Multicore Key-Value Stores. *PVLDB* 13, 7 (2020), 1091–1104.

[26] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. 302–313.

[27] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *PVLDB* 13, 7 (2020), 1078–1090.

[28] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

[29] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SIGOPS SOSP*. 456–471.

[30] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. In *DaMoN @ SIGMOD*. 8:1–8:7.

[31] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*. 371–386.

[32] Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. 2015. GRAPHITE: an extensible graph traversal framework for relational database management systems. In *SSDBM*. 29:1–29:12.

[33] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*.

[34] Andrew Pavlo and Matthew Aslett. 2016. What's Really New with NewSQL? *SIGMOD* 45, 2 (2016), 45–55.

[35] Holger Pirk, Oscar R. Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB* 9, 14 (2016), 1707–1718.

[36] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *DBPL*. 1–10.

[37] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The Graph Story of the SAP HANA Database. In *BTW*. 403–420.

[38] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, Muhammad Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD*. 979–990.

[39] Steve Scargall. 2020. *PMDK Internals: Important Algorithms and Data Structures*. Apress, 313–331.

[40] Steve Scargall. 2020. *Programming Persistent Memory*. Apress.

[41] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD*. 1907–1922.

[42] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *DaMoN @ SIGMOD*. 12:1–12:7.

[43] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *USENIX FAST*. 61–75.

[44] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *ICDE*. 461–472.

[45] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*. 473–488.

[46] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB* 10, 7 (2017), 781–792.

[47] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *USENIX ATC*. 349–362.

[48] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *USENIX FAST*. 169–182.

[49] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *PVLDB* 13, 4 (2019), 421–434.

Fixing Wikipedia Interlinks Using Revision History Patterns

Tova Milo
Tel Aviv University
milo@post.tau.ac.il

Slava Novgorodov
eBay Research
snovgorodov@ebay.com

Kathy Razmadze
Tel Aviv University
kathyr@mail.tau.ac.il

ABSTRACT

Wikipedia, the web-based free content encyclopedia project, is one of the most popular websites on the Web. Its “open-door" policy, allowing anyone to edit, has made Wikipedia the largest and possibly the best encyclopedia in the world. At the same time, the continuously evolving content, constantly updated by a large number of uncoordinated users, renders the maintenance of a clean, consistent encyclopedia an extremely challenging task.

The goal of the WiClean (WC) system presented in this paper is to assist Wikipedia editors in this difficult task. Specifically, we focus on the correctness of Wikipedia *inter-links* that point from one article (entity) to another. Such inter-links form a key component of the structured part of Wikipedia and their correctness is critical for coherent browsing. Given an entity type of interest, our highly parallelizable algorithm identifies relevant edit patterns across revision histories of Wikipedia entities of related types, along with time windows in which partial edits are tolerable. The discovered patterns/windows are then used by WC to alert Wikipedia editors on past edits that appear to be incomplete, as well as to provide users with on-line assistance as they update the encyclopedia. Our experiments with real-life Wikipedia data demonstrate the efficiency and effectiveness of WC in identifying actual errors in a variety of Wikipedia entity types.

1 INTRODUCTION

Wikipedia, the free-content web encyclopedia, is one of the most popular websites on the Web. Per Time magazine, Wikipedia’s "open-door" policy of allowing anyone to edit the data, has made it the largest, and possibly best, encyclopedia in the world [2]. Nonetheless, the continuously evolving content, constantly updated by a large number of uncoordinated users, renders the maintenance of a clean, consistent encyclopedia an extremely challenging task. To understand the volume of the updates, the English Wikipedia in 2018 consisted of 6 million articles, with an average of 3.4 million edits per month, by roughly 30K active editors [4].

The goal of our work is to assist Wikipedia editors in this difficult task. Specifically, we focus here on the correctness of *inter-links* that point from one article to another in the *structured* sections of Wikipedia (such as infoboxes and tables), which is critical for coherent browsing. Maintaining the integrity of these links is challenging, as illustrated by the following example.

Example 1.1. Consider the Wikipedia page of the soccer player Neymar. The links in its infobox point to the page of his current club, Paris Saint Germain F.C. (PSG), his place of birth, and so on. When Neymar moved to PSG in 2017, leaving his previous team, Barcelona F.C., the three related pages, *Neymar*, *PSG*, and *Barcelona F.C.* had to be updated.

There are three typical causes for inconsistently updating these links. First, Wikipedia editors are not provided with a comprehensive list of links that need to be updated as a result of such an event. A typical error related to player transfers is updating only the page of the new club and neglecting to update the page of the old club, which still incorrectly links to the player.

Second, different pages are often edited by different people, typically, in an uncoordinated manner. It could be that, e.g., the page of the club is updated by one dedicated editor, whereas no editor has taken up the responsibility of updating Neymar’s page or even noticed the absence of a corrected link. Moreover, no mechanism alerts the active editors of Neymar’s page of a related update, that may require action on their part.

Third, it is often impractical to correct all links simultaneously. For example, player transfers occur during predetermined periods, referred to as transfer windows, and tend to take a long time to be officially confirmed. In the meantime, many rumors regarding conflicting transfer destinations are posted in various media outlets. Consequently, in that span, there may be hundreds of edits of player pages, adding/removing new/old links, and reverting previous edits, whereas the club pages are commonly updated only once the transfer is officially approved.

More generally, Wikipedia contains very noisy data, as it could be edited by anyone, including bots[1], inexperienced editors, and opposite-agenda editors[2], resulting in editing conflicts[3] and dispute resolution[4]. This process of frequent conflicting edits, culminating in a consistent state, is a naturally evolving mechanism to mitigate noise, due to the distributed and asynchronous nature of Wikipedia edits. Thus, enforcing immediate corresponding updates to all relevant links during the dispute period is impractical and counterproductive. Moreover, the existence of a time window, that may range from hours to months (depending on the context of the update and the involved entities), during which partial inconsistent edits are tolerable, beyond serving as a necessary trigger for the dispute resolution process, also has the advantage of providing users with the most up-to-date, albeit tentative, information.

Previous work. Much research has been devoted to aspects of this problem in the more general context of detecting errors in *knowledge bases* (KBs) [22]. Some of these works [27, 30] also evaluated their solutions over Wikipedia, representing a snapshot of it as a KB, with pages as entities, and entity relations derived from inter-links. Over this representation and an input set of *integrity constraints*, pertaining to entity relations, the objective is to detect all their violations. While these works provide satisfactory solutions for the intended problem over KBs, casting the special case of inconsistencies in the constantly-evolving Wikipedia’s links into this generic framework, omits important practical considerations specific to the operation of Wikipedia.

To illustrate, continuing with our example of player transfers, a possible constraint over the corresponding KB may state that if player A links to club B, then club B also links to player A and vice versa. If there exists only one link or two contradictory links, then a violation of this constraint is detected. There are several drawbacks to applying this approach as a comprehensive solution.

[1] https://www.bbc.com/news/magazine-18892510
[2] https://en.wikipedia.org/wiki/Wikipedia:Lamest_edit_wars
[3] https://en.wikipedia.org/wiki/Edit_conflict
[4] https://en.wikipedia.org/wiki/Wikipedia:Dispute_resolution

First, the most crucial drawback relates to the fact that the constraints are static and lack any temporal dimension. Concretely, the constraint does not account for the time window, discussed in Example 1.1, during which partial edits are acceptable (and, in some cases, practically unavoidable). An inconsistency should be resolved at the earliest appropriate moment but not earlier. In this case, this earliest moment is arguably the end of the transfer window. However, detecting the constraint violation right after Neymar's page is linked to PSG, without a link in the other direction, is treated the same as detecting it long after the end of the transfer window. Consequently, one is uncertain whether to take immediate action or allow the unsupervised process of sequential refining edits to run its course and converge into a consistent state. We note that solutions suggested in previous works (e.g., [27]), discussed above, can successfully identify 'window-less' edits' combinations. That is edits that should all be applied simultaneously and are distributed uniformly across the timeline of Wikipedia's revision history. However, our empirical analysis identified many edit patterns associated with specific time windows, such as in Example 1.1. Our work is, thus, complementary to the above works, as it aims to address specifically these patterns.

Second, most works assume the existence of a set of constraints as input for the solution framework. This is not realistic in the case of Wikipedia. Wikipedia entries encompass a wide array of domains and sub-domains, each with its own set of constraints. While there are broad similarities across related domains, each domain may be infinitely nuanced. Given the volume of domains, entity types, and case-specific subtleties, the task of comprising a nearly exhaustive list of important constraints is impractical, particularly if one is also interested in complex relations (where, e.g., a combination of 10 pages must be consistently updated).

In this line of research, closest to us, is the recent work of [36], where the focus is on Wikipedia, and on top of detecting violations of the given constraints, the solution produces corresponding *correction rules*, that dictate how one can resolve partial edits. This is inferred by examining the revision history, identifying the most common patterns of revision actions for completing each type of partial editing. Nevertheless, this work also does not aim to identify tolerable time windows and targets the scenario where the list of constraints is provided as input (a more detailed comparison to this and previous works is presented in Section 2).

Our approach. To address the above limitations, we present in this paper WiClean (WC), a system that automatically infers common edit patterns (combinations of edits), along with a time window for allowing partial edits of each pattern, alerts editors of inconsistencies, and suggests concrete corrections.

The thesis underlying WC is that the majority of Wikipedia updates follow desirable patterns and lead to consistent states. WC, thus, mines revision logs to identify common update patterns and the time windows in which they occur. Potential errors are then detected by updates that deviate from the patterns and are not completed within the corresponding window. For such partial patterns, WC suggests all completions to known patterns, providing statistical metadata to facilitate an informed course of action.

Before describing our solution techniques, we illustrate the format of the revision history. Figure 1 depicts excerpts from the revision histories of players and clubs merged into a single timeline. The Subject column identifies the article where the addition/removal of a link occurred, the Object column identifies the article to which the added/deleted links point, and the Relation column describes the link type (the column R will be explained later). One can see that, after several edits and reverts, the transfer of Neymar is reflected in his and the teams' pages.

#	+/-	Subject	Relation	Object	Time	R
1	-	Neymar	current_club	Barcelona F.C	...1531	0
2	-	Gianluigi Buffon	current_club	Juventus F.C.	...1534	1
3	-	Neymar	in_league	La Liga	...8711	1
4	-	Barcelona F.C	squad	Neymar	...2804	0
5	+	Neymar	current_club	PSG F.C.	...3321	0
6	+	PSG F.C.	squad	Neymar	...8263	1
7	+	Barcelona F.C.	squad	Neymar	...4040	0
8	+	PSG F.C.	squad	Gianluigi Buffon	...4051	1
9	+	Gianluigi Buffon	in_league	Ligue 1	...3330	1
10	+	Neymar	in_league	Ligue 1	...8711	1
11	-	Juventus F.C.	squad	Gianluigi Buffon	...4058	1
12	+	Neymar	current_club	Barcelona F.C	...5861	0
13	-	Kylian Mbappe	current_club	Monaco F.C.	...9459	1
14	-	Neymar	current_club	PSG F.C.	...3732	0
15	-	Gianluigi Buffon	in_league	Serie A	...3380	1
16	-	Neymar	current_club	Barcelona F.C	...6109	1
17	+	Neymar	current_club	PSG F.C	...7694	1
18	-	Barcelona F.C.	squad	Neymar	...8001	1
19	+	Kylian Mbappe	current_club	PSG F.C.	...9589	1
20	-	Monaco F.C.	squad	Kylian Mbappe	...9451	1
21	+	PSG F.C.	squad	Kylian Mbappe	...9885	1

Figure 1: Actions from revision history of several articles

Methods. Formally, we model Wikipedia entities (articles) and the links between them as a graph. Nodes and edges are labeled by type names. Intuitively, the revision history of each article records the edits made to the outgoing links of the corresponding graph node. Given an entity type of interest, our algorithm identifies meaningful relevant edit patterns across revision histories, along with time windows in which partial edits are acceptable. By making an analogy between link edits (resp. edit patterns) and graphs (graph patterns), we can harness conventional graph mining algorithms to our context. However, some important adaptations must be made to account for (1) the Wikipedia type hierarchy[5] that requires the examination of a larger number of potential patterns, and (2) the distributed nature of the edits across revision histories of multiple entities, that makes the construction of the full edits graph prohibitively expensive. For the former we introduce a join-based computation (optimized by the underlying SQL engine) to quickly prune infrequent patterns; for the latter we use incremental graph construction that considers only relevant entity types.

The discovered windows and patterns are then used by WC to assist Wikipedia editors in correcting/updating Wikipedia links. Here again, we employ an optimized join-based computation to quickly identify potential errors. WC both alerts Wikipedia editors on past edits that appear to be incomplete as well as provides users with on-line assistance as they update Wikipedia.

Our contributions can be summarized as follows:

- **Model.** We formulate and present a simple, natural model for capturing time windows and update patterns of interest. Given an entity type t, our goal is to find related and common update patterns across the Wikipedia graph. Such updates may involve entities of the same or other types. We first introduce the notion of *abstract* update actions that generalize a set of actions involving specific entities to general patterns over the corresponding entity types. We then define the notion of *connected patterns* which include abstract actions that are related (possibly transitively) to entities of the input type of interest. The *frequency* of a pattern, within a time frame w, is then naturally defined as the fraction of entities of type t that participate in a pattern that occurs within the time frame w (Section 3).
- **Identifying windows and patterns.** Building on algorithms for graph mining, we devise a scalable, highly parallelizable algorithm, based on the following three points. (1) We represent

[5]Typically around eight hierarchy levels.

the identified patterns by relational tables, incrementally computed by dedicated relational queries. This allows harnessing the effective optimizations of the SQL engine underlying WC. (2) Unlike conventional graph mining algorithms that assume that the entire graph is given as input, our focus on connected patterns allows WC to incrementally consider only the entity types (and their corresponding revision histories) that may potentially be related to the input type via frequent edit patterns, thereby significantly saving on graph construction. (3) We focus on non-overlapping time windows and split the revision histories accordingly. This reduces the number of actions (edits) to be considered for each window (and resp. the size of the edits graph) and allows parallelizing the processing of the action sets in the different windows (Section 4).

- **Using Windows and patterns.** An immediate application of the discovered patterns is to alert Wikipedia editors on partial edits performed in past windows, as well as to assist users in current edits. For that, we examine the discovered windows and then signal, for each window and pattern, partial edits that may be extended to a full pattern occurrence. Our algorithm builds on the previously mentioned relational representation of patterns and employs dedicated outer-join queries to identify partial pattern occurrences (Section 5).
- **Implementation and experiments.** We have implemented our solution and employed it over real Wikipedia data. We considered a variety of Wikipedia entities, identifying a multitude of interesting time frames and corresponding relevant frequent edit patterns, and signals of updates that deviate from the mined patterns. Our experiments demonstrate the effectiveness of our approach for identifying real-life errors. The experiments further demonstrate the efficiency and scalability of our algorithms, compared to competing baselines (Section 6).

To complete the outline of the paper, we overview related work in Section 2 and discuss future work in Section 7.

Finally, we note that the prototype of WC was demonstrated in [20]. The short paper accompanying the demonstration provided only a high-level overview of its capabilities and user interface whereas the present paper details the model and algorithms underlying our solution as well as their experimental evaluation.

2 RELATED WORK

We overview related work from several related fields.

Wikipedia Cleaning. Much effort has been devoted over the past years to the cleaning and correction of errors in Wikipedia. Our work, which focuses on link correction, is complementary to works on entity resolution, completeness prediction, and vandalism detection [9, 33]. Similarly to our work, [13] also aims to improve inconsistencies in Wikipedia's infoboxes, representing it as an RDF database. However, [13] does not take the revision history into account and instead uses user interaction as the main tool. In contrast, our algorithm requires no user assistance, other than setting the initial parameters.

Revision history as a tool. Revision histories have been used in multiple areas, e.g., in program repairing, in recording provenance in knowledge bases and assisting query answering [10]. In Wikipedia, revision histories have been leveraged for various purposes, such as the discovery of controversial topics, the estimation of an article's translation quality and the detection of vandalism [23]. Other lines of work attempt to learn how to use the edits to enrich Wikipedia, e.g. to edit infoboxes with news extracted from tweets, or to connect Wikipedia edits to recent news articles [17]. Our work is complementary to these efforts, considering the consistency/completeness of edits to multiple related entities.

As mentioned in the introduction, particularly close to our work is [36], which, similarly to WC, infers from edit histories in Wikidata knowledge bases how to correct inconsistencies/violations. Nevertheless, this problem is formalized over a different model with similar but different objectives. One important difference is that in the setting of [36], *the constraints are provided in advance*, and the focus is on correction rules (for violations of these constraints) mining from relevant past edits. Whereas, one of the key contributions of our work is the derivation of such constraints (edit patterns, in our context). Moreover, the setting of [36] does not take into account the time frames in which a given constraint should or should not be enforced. Another key difference is that [36] do not harness the Wikipedia type hierarchy as a means to enrich their constraints of correction rules.

Other works that use the Infobox revision history focus on cleaning tasks. These include refining infobox titles by locating duplicate attributes within each entity type, predicting when a given infobox is likely to be updated and by whom, and identification of vandalizing editors [8].

Constraints inference and enforcement. The patterns we identify can be viewed as a form of *integrity constraints*. There is a large body of work devoted to inferring and enforcing such constraints. Two recent examples are [27, 30]. In [30], both positive and negative examples are used to infer the constraints. Their approach consists of greedily identifying, at each step, the most promising rule, in terms of the coverage of the positive examples. As [30] focus on identifying rules that make good predictions, some of the rules that exceed the confidence threshold will not be found. An alternative approach is taken by [27], where rules are mined via an exhaustive, breadth-first search method. They devise sophisticated pruning strategies and optimizations that enable their solution to efficiently run on large KBs, such as Wikidata.

Many other approaches to KB correction have been explored in the literature, e.g., discovering *denial constraints* [14] and error detection via the few-shot learning framework (e.g., [22]).

A key difference, in our setting, is that the constraints need to be enforced only outside the time frames in which inconsistencies are acceptable. Thus, we focus on a different objective, where in addition to the update patterns, we also identify the corresponding time window for each pattern. Another difference is that we identify patterns from the sequence of actions in the revision log, and not from a static snapshot of the knowledge base. Moreover, the above works (barring [36], which we discussed separately above), are concerned with detecting the rules/constraints, while one additional objective, in our setting, is to also compute correction suggestions for violations (partial patterns) of these rules. Lastly, to our knowledge, we are the first to leverage the type hierarchy to consider more nuanced rules, at varying levels of abstraction.

The importance of considering consistency, w.r.t. a sequence of actions, has recently been emphasized in the vision paper of [12]. Our work matches their motivating use-case, which advocates the usage of Wikipedia revision logs for data cleaning. Another related, complementary line of work deals with optimizing the corrections procedure over the detected constraint violations [11, 19]. It would be interesting to examine whether their techniques may be employed in our setting, to further optimize WC.

(Sequential) itemset/association rule mining. Algorithms for frequent itemset/association rules mining have been the focus of many works, including contexts where the mined items belong to a type hierarchy [32].As we seek connected patterns, conventional

a-priori style algorithms for frequent itemsets mining [7] inapplicable to our setting. Such algorithms recursively assemble larger frequent itemsets from smaller ones, but arbitrary sub-patterns of a connected pattern may not be connected, w.r.t. the input type. Consequently, our solution exploits principles from graph mining algorithms rather than general frequent itemsets.

Another closely related line of work deals with *sequential* itemset/association rules mining, where the pattern is mined from a sequence of items [34, 41] and [5] which discovers temporal rules for web data cleaning. In these works, the focus is also typically on arbitrary items set, rather than connected patterns. More importantly, the order of the items in the sequence is important in these works. In contrast, as explained in Section 3, in our case, only the co-occurrence of items within the given window matters, whereas their relative positioning within the window does not.

An interesting set of works that deal with sequential patterns mining studies probabilistic or uncertain databases [18, 29]. In our setting, there is inherent uncertainly, w.r.t. the (in)correctness of the identified partial updates, and thus examining the connection to such works is an interesting direction for future research.

Graph mining. Graph mining algorithms (see survey in [24]) can be roughly divided into two categories: algorithms that mine patterns in a set of graphs (e.g. [39]) and algorithms that are provided with a single large graph (e.g. [26, 28]). Our context is the latter. Multiple notions of graph pattern frequency have been proposed in the literature, many of which consider the number of distinct isomorphisms from the given pattern graph to the input graph [15, 24]. However, as our goal is to characterize how frequent a pattern is *relative to a particular entity type of interest*, we employ here the notion of frequency, inspired by [16], that counts the number of nodes, out of all nodes of the given type, that are involved in some pattern occurrence. Our notion can also be viewed as a special case of the MNI support in [15], where the isomorphism count focuses only on the given entity type.

As discussed above for association rules mining, since our focus is on connected patterns (and the corresponding frequency notion), algorithms that consider arbitrary sub-graphs (e.g. [21]) are unsuitable for our setting. We follow instead the "grow and store" approach of [26], that iteratively expands previously identified (connected) patterns. However, two issues must be addressed when adapting such a scheme to our context. First, the need to support the Wikipedia type hierarchy entails a richer order relation among patterns (see Section 3), which, to our knowledge, is not supported by any of the existing algorithms for mining connected patterns in graphs. Second, [26] (and all other comparable works), assume that the algorithm receives as input the entire graph. This is impractical in our context. Specifically, as our experiments demonstrate, materializing the complete edits graph from a massive number of entity revision histories is infeasible. Our dedicated algorithm addresses both these issues. In general, modifying solutions that expect the entire graph as input, is, arguably, not trivial. For instance, the work of [40], which leverages the embedding of the nodes to mine patterns, cannot be straightforwardly integrated into our approach of gradually examining larger subgraphs, as the embedding loses its utility if the underlying graph changes.

Another related line of work is Link Prediction [35, 38] that discover missing links within Wikipedia. However, these works do not detect incorrect links that should be removed.

Wikipedia information extraction. To conclude, we note that one may think of the patterns/time windows that we derive as a particular type of information, extracted from Wikipedia revision logs. Much previous work has been devoted to information extraction from Wikipedia *articles* (e.g. [31]) rather than their edit history. As mentioned, some works consider the revision logs, but for other purposes, and could be useful information or tool for us. In particular, [37] devise optimization methods for processing Wikipedia's revision history, as it is a massive and complicated dataset, and [25] infers the level of expertise of a specific editor from statistics of conflicts with other editors.

3 PRELIMINARIES

We start by presenting the data model underlying the system.

Wikipedia Graph. We model the relations between entities at a given point in time using a graph $G(V, E)$. Each node represents an entity and is labeled by a unique name (e.g. Neymar) and a type (e.g. *soccer player*). Each edge represents a relationship between two entities and is labeled accordingly (e.g. *current_club*).

We use an alignment from Wikipedia entities to DBPedia [1] to derive the entity types. The link labels (relationship names) are derived directly from Wikipedia. In general, the types belong to type taxonomy - the higher the type is in the taxonomy the more general it is - and an entity may have multiple types. For two types t, t' we use $t' \leq t$ to denote the fact that t either equals to t' or generalizes it. For example, $Soccer\_Player \leq Athlete \leq Person$. We assume that each entity e has one most specific type to which it belongs and use it as its label, denoted $type(e)$. For a type t we use $entities(t)$ to refer to all entities labeled by a type $t' \leq t$.

Actions and inverse actions. The revision history of Wikipedia entities contains edits to the graph edges. We particularly consider two types of actions: adding *new* edges and deleting *existing* ones. Our model associates each action with a time stamp. We use a triplet of the form $a = (+, (u, l, v), t)$ (resp. $a = (-, (u, l, v), t)$) to denote the addition (rep. deletion) of edge from u to v with label l at time t. We use $source(a) = u$ and $target(a) = v$ to denote the source and target entities, resp., of the added/deleted edge. We say that an action a' is the *inverse* of a preceding action a, denoted $a' = Inv(a)$ if applying a' after a leaves the graph unchanged.

For example, in row #1 in Figure 1 we see an update to Neymar's Wikipedia entry, when a user removed ($-$) the Barcelona ($= v$) team that Neymar ($= u$) was playing at ($= l$), at a certain time (=t), and, action #12 is an inverse action of action #1.

Note that, in Wikipedia, each action appears at the revision history of the *source* node of the edge. Intuitively, this is because the revision history of each article records the edits made to the outgoing links of the corresponding graph node. Updates of other incoming links are recorded in the revision logs of these other pointing entities. Continuing the above example, the two actions #1 and #3 will appear in the revision history of Neymar's page, and the gray action set in Figure 1 is the set of actions taken from entities of the same type (soccer_player).

(Reduced) set of actions. Given a Wikipedia graph $G(V, E)$, a set of entities $S \subseteq V$, and a time frame (referred to as *window*), we consider the set of all actions (denoted as A) that were recorded in the revision history of the entities in S, within the given window.

For instance, Figure 1 shows the set of actions recorded in the revision histories of the entities S = *{Neymar, Kyian_Mbappe, Barcelona_F.C., Gianluigi_Buffon, PSG_F.C., Monaco_F.C., Juvetus_F.C.}* at a given time frame. Observe that all the updated links are *outgoing* links from the entities in S.

In the update processes, some edits may naturally be reversed. To consider only the final effect we focus on *reduced actions sets* that do not include action and its inverse. More formally, given a graph G, we say that two action sets are *equivalent* if, when the

actions are applied on G in the order of their timestamps, they yield the same graph. The reduced set of actions, that remain by removing the rows that their value in column R equals to 0 in the table of Figure 1. We denote it as reduced actions from Figure 1.

Note that up to possibly different timestamps, the reduced version obtained through this iterative removal process is unique, as it contains the same set of graph update operations. Furthermore, the timestamps are no longer important as any permutation of the actions yields the same output graph. We thus consider from now on only reduced sets of actions and ignore the timestamps, referring to actions as pairs $a = (op, (u, l, v))$ where $op \in \{+, -\}$.

Abstract actions. Since we are trying to find general update patterns across the Wikipedia graph, we want to generalize a set of actions involving specific entities to general patterns over the corresponding entity types. For that we define the notion of *abstract actions*. We associate with each entity type t an infinite set of variables $t_1, t_2, \ldots$. Then, an *abstract action* is the pair of the form $a = (op, (t', l, t''))$ where $op \in \{+, -\}$, t' and t'' are type variables, l is an edge label.

Patterns. We define a *pattern* as a set of abstract actions. We consider two patterns identical if they are the same up to isomorphism on the variable names of the same type. We refer to a pattern that contains only a single action as a *singleton* pattern. Given a pattern p we say that a set A' of concrete actions is a *realization* of p (resp. that p is an abstraction of A') if A' may be obtained from p by replacing each variable of type t by a some Wikipedia graph node in $entities(t)$, s.t. distinct variables are assigned different Wikipedia graph nodes.

An observation that will be useful in the sequel is that for a given action a, the set of its possible abstractions can be easily computed by traversing the type hierarchy and replacing $source(a)$ (resp. $target(a)$) by some variable of type $\geq type(source(a))$ ($\geq type(target(a))$).

To illustrate the above notions, in the reduced actions in Figure 1 lines #2 and #13 are both realization of the singleton pattern {-, (player1, current_club, team1)} (which we consider identical, e.g., to the isomorphic pattern {-, (player2, current_club, team2)}). On the other hand, the reduced actions in Figure 1 contain no realization of the pattern
$[\{-, (player1, current\_club, team1)\},$
$\{-, (player1, current\_club, team2)\}]$
as the assigned team nodes have to be distinct in the realization, but all players in the table were removed from a single team.

(Abstract) actions graph. It is useful to make an analogy between action sets and graphs. Given a set of concrete (resp. abstract) actions A (p), consider the directed labeled graph g_A (g_p) with a node per each entity in A (variable in p), labeled by the entity (variable) type name, and where there exists an edge from node v_1 to v_2, labeled $[op, l]$, iff A (p) includes an (abstract) action of the form $(op, (v_1, l, v_2))$. We refer to these graphs as abstract graphs as the actual entity identities (resp. the variable names) that the nodes represent are insignificant.

With this graph view, a realization of a pattern p in a set of actions A corresponds to an isomorphism from g_p to a subgraph of g_A, where the type of each node in p either equals or is more general than the type of its corresponding node in g_A. Given a type t, we say that the pattern p is **connected** (w.r.t. t) iff g_p contains a node variable of type t from which all other nodes are reachable.

Connected patterns. Given an entity type t, we are interested in entities' updates that are related (possibly transitively) to entities of type t. We thus focus on *connected* patterns, where the updated edges are related.

Definition 3.1. For an update pattern p, let g_p be its corresponding abstract graph. Given a type t, we say that the pattern p is **connected** (w.r.t. t) iff g_p contains a node of type t from which all other nodes are reachable.

In the discussion below we refer to such a node (variable) as the pattern's **source** (w.r.t. t). If multiple such nodes exist in the graph, we arbitrarily pick one to serve as the pattern's distinguished source and we use below the term source to refer to this single distinguished node, and denote is as $source_t$.

For example, the pattern shown in Figure 3 is connected w.r.t. to the type $player$. Its corresponding graph g_p appears in Figure 2(a) where all nodes are reachable from the source node $player\_1$. But if we replace the variable $player_1$ in lines 11 and 13 of Figure 3 by a new variable $player_2$, then the pattern becomes disconnected, see Figure 2(b), and composed of two smaller, connected patterns - the abstract actions in lines 10, 5, 2, 7 (with source $player\_1$) and the abstract actions in lines 11, 13 (with source $player\_2$).

For a type t we only consider patterns that are connected w.r.t. t. Thus, for brevity, we use below the term *pattern* to refer to a connected pattern, and omit the type t when clear from the context.

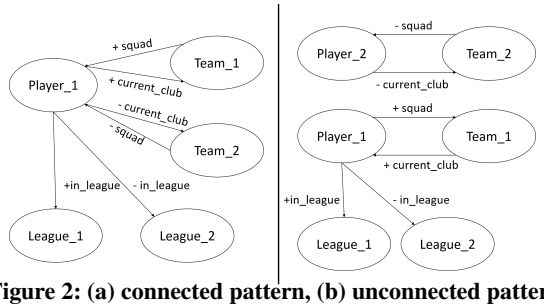


Figure 2: (a) connected pattern, (b) unconnected pattern

Frequent patterns. To define that a pattern is frequent we would like to measure the amount of support that a pattern has, regarding the seed type entities.

Many notions of patterns frequency have been considered in the graph mining literature. Common notions consider the number of distinct isomorphisms from the given pattern graph to the input graph (e.g. occurrence-based support [24] and MNI support [15]. However, as our goal is to characterize how frequent a pattern is *in the context a particular seed type of interest*, we employ here a notion of frequency inspired by the [16] that counts the number of nodes (out of all nodes of the given seed type) that are involved in some pattern. For readers familiar with the *MNI*-based support in [15], we note that our notion of frequency can also be viewed as a special case where the isomorphism count focuses only on the seed entity type node.

Given a type t, a pattern p and a set A of actions, we define the *frequency* of p (w.r.t. to t and A) as the fraction of entities of t that participate as source nodes in a realization of p is A.

Definition 3.2. The *frequency* of a pattern p in a set of actions A, w.r.t. to a type t in p, is defined as $frequency(p, A, t) =$

$$\frac{|\{e \in entities(t) \mid e \text{ appears in realization } entities(t)|}{|entities(t)|}$$

To continue with our running example, consider the actions in Figure 1 and the pattern in Figure 3, and assume there are overall five players in Wikipedia. The frequency of this pattern in the given actions set, w.r.t. to the type $player$, is 0.2 because there is only one player (Neymar) that the patterns hold for (with Neymar mapped to $player\_1$), out of the five existing players. However, the frequency of the partial pattern displayed in figure 3 in lines 1

#	Edit type	Subject	Relation	Object
10	+	$player_1$	current_club	$team_1$
11	-	$player_1$	current_club	$team_2$
5	+	$team_1$	squad	$player_1$
13	-	$team_2$	squad	$player_1$
2	+	$player_1$	in_league	$league_1$
7	-	$player_1$	in_league	$league_2$

Figure 3: Pattern found from set of action in Figure 1

and 2 (gray lines) in this actions set (again w.r.t. the type *player*), is 0.4 because there are 2 players for which that pattern holds.

Partial Order of Patterns. Given a type t, a set A of actions and frequency threshold τ we will be interested in finding patterns whose frequency in A (w.r.t. the given type) is above the threshold. To avoid redundancy, we would like to consider only the most *specific* such patterns. Formally, we say that a pattern p is more specific than a pattern p' (alternatively, p' is more general than p), denoted $p \prec p'$, if p' may be obtained from p by removing some abstract actions, replacing some type variables in p by corresponding variables of a more general type, or both. An alternative definition is close frequent sub-graph, as defined in [39]. To illustrate, for the patterns:

$p_1 = \{(+, (player_1, current\_club, team_1)),$
$(-, (player_1, current\_club, team_2))\}$
$p_2 = \{(+, (athlete_1, current\_club, team_1)),$
$(-, (athlete_1, current\_club, team_2))\}$
$p_3 = \{(+, (athlete_1, current\_club, team_1))\}$

we have that $p_1 \prec p_2 \prec p_3$.

Thus, given a type t and a set A of actions our goal will be to find the most specific patterns with a frequency above a given threshold. Our formal definition refines the closed frequent graph pattern notion of [39], taking the type hierarchy also into consideration when ordering patterns.

Definition 3.3. Given a set of actions A, a type t, and a frequency threshold τ, we say that a pattern p is a *most specific frequent pattern* in A (w.r.t. t and τ), if $frequency(p, A, t) \geq \tau$ and there is no pattern $\hat{p} \prec p$ where $frequency(\hat{p}, A, t_i) \geq \tau$.

Relatively frequent patterns. Finally, note that in the discussion so far, the frequency of patterns p was measured w.r.t. a given type, as the percentage of entities of the given type that serve as a pattern source. In some cases, it is interesting to further explore what percentage of these entities adhere to a more specific pattern p'. For example, what percentage of players among the ones that move to a new team also, change the league. For that we define the notions of *relative* frequency and *relative* frequent patterns.

Definition 3.4. For two patterns p, p' s.t. $p' \prec p$, the *relative frequency* of p' w.r.t. p in a set of actions A (for a given type variable t), is defined as

$rel\_frequency(p', p, A, t) = \frac{frequency(p', A, t)}{frequency(p, A, t)}$.

Definition 3.5. Given a set of actions A, a type t, a pattern p and a relative frequency threshold τ_{rel}, we say that a pattern p' is a *most specific relative frequent pattern* in A, w.r.t. t and p, if $rel\_frequency(p', p, A, t) \geq \tau_{rel}$ and there is no more specific pattern $\hat{p} \prec p'$ where $frequency(\hat{p}, p, A, t) \geq \tau$.

4 FINDING WINDOWS AND PATTERNS

Intuitively, given an entity type t of interest, we wish to signal out significant time frames and identify the most specific frequent patterns in them.

We will first explain how, given a specific window w and frequency threshold τ, the most specific frequent patterns in w (w.r.t. type t), are efficiently identified. The extraction of relative frequent patterns is similar. Finally, we will explain how the windows and thresholds to examine are selected.

As noted in Section 3, the (reduced) set of edit actions performed over Wikipedia entities in the time window w may be viewed as a graph. Thus one may harness graph mining algorithms, such as the ones presented in [24] to identify frequent connected patterns. Such algorithms work roughly as follows. Starting from patterns consisting of a single edge, they incrementally expend the patterns with new edges. At each iteration, they check which of the extended obtained patterns are frequent, prune all the others, and iteratively continue expending the frequent ones.

There are, however, two important issues that one has to address when adapting such a scheme to our context.

1. Supporting the Wikipedia type hierarchy entails a richer order relation among patterns (as defined in Section 3), which to our knowledge is not supported by any of the existing algorithms for mining connected patterns in graphs. While the modifications to the algorithms, to support this, are rather immediate, the number of patterns that now need to be examined becomes larger, and thus the patterns' frequency test must be performed efficiently. For that, we represent each graph's type of relation as a relational table, containing its pattern realizations. That allows us to utilize a join-based computation (optimized by the underlying SQL engine) to quickly prune infrequent patterns.

2. Observe that common graph mining algorithms assume that a full graph is given as input to the algorithm. In our setting, the revision histories are distributed across all Wikipedia entities, and (even when restricted to the time window w) their overall size can be very large. Thus, as our experiments show, materializing the full graph that represents them may be prohibitively expensive. To avoid this, we embed into the discovery of the incremental patterns an analogous incremental graph construction, that materializes only revision histories of entity types that may potentially be related to the input type t via frequent edit patterns. Our pattern mining algorithms is detailed in Algorithm 1. For better understanding the pseudo-code, we first outline the data structures and notations that we make use of. For space constraints, an illustrative example appears in our technical report [3].

4.1 Data Structures and Notation

For each considered time window w, the algorithm incrementally extracts, from the revision histories determined to be relevant, the set of actions performed within the time frame. The actions are abstracted and stored in a dictionary called $abstract\_actions$ whose keys are the time windows. Thus, $abstract\_actions[w]$ denotes a set of abstract actions with realizations in the window w. The corresponding realizations of each such abstract action are stored in a dictionary called $realizations$ whose keys are the time frame and abstract action. Thus, $realizations[w][a]$ denotes the set of realizations of abstract action a within window w.

The identified (relative) patterns, for each time window w (and pattern p in w), are stored in a dictionary named $patterns$ (resp. $rel\_patterns$) whose keys, again, are the time frames (and related patterns). Thus $patterns[w]$ (resp. $rel\_patterns[w][p]$) denotes the set of (relative) patterns computed for time window w (and pattern p in w). We overload notation and also use below $realizations[w][p]$ (resp. $realizations[w][p][p']$ to denote the realizations of the (relative) pattern p (p') within time window w. As mentioned above, the pattern realizations are implemented as relational tables. We will explain this point in details below. Finally, we use an auxiliary data structure $tested[w]$, whose keys are the time frames, to record partial patterns that have already been examined in the computation for the window w.

4.2 Pattern mining

We are now ready to present the Algorithm 1. As mentioned above, the algorithm follows the line of graph mining algorithms such as [15], starting from singleton patterns and incrementally expending them. While doing so it incorporates into the processing the two optimizations mentioned above, to ensure efficient processing in our particular setting. We note that several additional optimization techniques have been introduced in [24, 39], e.g. to minimize the used storage and search space. These are orthogonal to ours and thus, for simplicity of presentation, we follow below the basic scheme of the incremental pattern construction (to which these orthogonal optimizations can later be applied if desired).

Initialization. Our initial entity set S contains the entities of input type t. First, we extract for the given window w edit actions performed on entities in S in time window w. We reduce the set of actions, eliminating redundant edits and computing the possible action abstractions (as explained in Section 3) and store them in $abstract\_actions[w]$ and their corresponding realizations in $realization\_table[w]$. This is performed using the function $reduced\_and\_abstract\_actions(S, w)$ (line 1). $patterns[w]$ stores only abstract actions (singleton patterns) whose source is the seed type t and their frequency in w exceeds the threshold (line 2). We explain below how the frequency is efficiently computed.

Interleaving graph and patterns expansion. We next interleave the extension of considered entity set (and, correspondingly, the considered subgraph representing their respective revision histories), with the extension of the patterns.

To determine which other related entities (and, respectively, entity revision histories) should be considered, we examine the frequent patterns identified so far, to see which additional entity types appear in them, if any (line 4). Correspondingly, we add their (reduced) revision histories within w to the set of considered actions. For that, we employ again the function $reduced\_and\_abstract\_actions(S, w)$ (line 8) that reduces the revision histories and adds the actions abstraction (and their corresponding realizations) to $abstract\_actions[w]$ (resp. $realization\_table[w]$).

Next, we iteratively consider for each previously discovered frequent pattern $p \in patterns[w]$, its graph g_p and attempt to extend it with additional edges (abstract action) $a \in abstract\_actions[w]$, that has not been considered for it yet (lines 9-14). The procedure uses the auxiliary global variable $tested[w]$, (initially the empty set) to record pairs of patterns and actions that have already been examined. It is important to note that by considering all action abstractions (rather than just their base type) we can construct patterns at all abstraction levels.

Extended patterns whose frequency exceeds the threshold are added to $patterns[w]$ (line 14). We will explain later how the pattern realizations and frequency are efficiently computed. When the frequent patterns can no longer be extended *w.r.t. the current set of abstract actions/action realizations*, we check again whether the discovered patterns contain new types whose actions have not yet been considered (line 4). If so, we repeat the graph and patterns extension (lines 5 - 15). Observe that the incremental nature of the patterns' construction allows refining the previously derived patterns with the newly added abstract actions, rather than computing frequent patterns from scratch. In other words, the extension of the actions graph, and the extension of the patterns (w.r.t. the extended graph), interleave well.

Note that, in the presentation so far we keep in $patterns[w]$ *all* the discovered frequent patterns and not just the most-specific ones. This is because such general patterns may still be useful, in later iterations, being expended to other, different most-specific patterns. However, an optimization that we can still employ here is the removal of these (not most-specific) patterns whose expansions have been fully examined, e.g. where all the entities types occurring in them have been thoroughly processed (line 15). Another optimization that we employ (omitted from the pseudo-code) is the cashing of the computed frequencies/realization tables, to be reused if the same patterns are later re-examined with different thresholds. When all patterns have been discovered, we select the most specific ones and return them (line 16).

Computing patterns realization and frequency. To complete the picture we need to explain how the patterns realizations and frequency are computed in lines 12-13 of the algorithm. To efficiently compute (and extend) pattern realizations, we represent each pattern realization in $realizations[w][p]$ by a *relational table* whose attribute names correspond to the pattern variables names, and whose tuples capture the different realizations of the pattern in the given time window (namely the qualifying assignments of concrete Wikipedia graph nodes to the pattern variables).

Now, note that given a pattern p and an abstract action a, there may be several ways to extend the graph g_p with a. First, a's source may be "glued" to any of the nodes (variables) in p of the same type as a (if such exist). Second, for each such possible gluing, a's target may either be added to the pattern as new pattern node (in which case g_p is extended by both a new edge and a new node) or the target may also be glued to an existing same type node (in which case g_p is extended by only a new edge).

We process each such possible extension as follows. Let p' be such an extended pattern. An important observation is that, using the relational representation discussed above, the realization table of the extended pattern p' can be easily computed, from $realizations[w][p]$ and $realizations[w][a]$, via a join-based query. For the glued pattern/action nodes we use equijoin on the corresponding attributes, whereas for the new node (if such exists), we require inequality to all same type attributes. Finally, we only need to project a single column for each pattern attribute. Then, the frequency of a pattern p w.r.t. a type t can be easily computed from the relation, by an SQL count operator that counts the number of distinct nodes appearing in the column corresponding to the pattern's source variable, (then dividing the count by the cardinality of $entities(t)$).

Mining Relative Patterns. To conclude, we note that the computation of *relative* frequent patterns proceeds in a similar manner. The only difference is that each pattern p we begin the expansion process starting from p itself, and relative frequency (rather than just frequency) is computed similarly, but using the formula in Definition 3.4. We omit the details for space constraints.

4.3 Finding Windows and Thresholds

So far we assumed that we are given a window w and a threshold τ, and our goal was to identify the (relative) frequent patterns in w, w.r.t. the seed type t. To identify windows and thresholds of potential interest, we use a simple heuristic, which our experiments show to be extremely effective.

We restrict our attention to non-overlapping time windows and split the revision histories accordingly. This allows parallelizing the processing of the action sets in the different windows. Our analysis of real Wikipedia data indicates this to be a reasonable design choice. For an input type t there are very few meaningful (update-wise) time frames that overlap and those can be merged into a somewhat longer window that includes both update patterns.

Algorithm 1: Mine connected patterns

Input: entity set S,Wikipedia type t, window w, frequency threshold τ, relative threshold τ_{rel}
Output: (relative) patterns and their time frames: $patterns[w]$, rel_patterns[w][p]

1 call reduced_and_abstract_actions(S, w) to create abstract_actions[w] and realizations[w];
2 patterns[w] = $\{\{a\} \mid a \in$ abstract_actions[w] $\wedge type(source(a)) = t \wedge$ frequency($\{a\}$) $\geq \tau$ $\}$;
3 tested[w]={};
4 **while** *new type names found in patterns[w]* **do**
5 **foreach** $p \in$ *patterns[w]* **do**
6 **foreach** *new type name* $t \in p$ **do**
7 S = get_entities(t);
8 call reduced_and_abstract_actions[S, w] to expand abstract_actions[w] and realizations[w];
9 **while** *there exists* $p \in$ *patterns[w],* $a \in abstract\_actions[w]$, *s.t.* $(p, a) \notin$ *tested[w]* **do**
10 tested[w]=tested[w] $\cup \{(p, a_i)\}$;
11 **foreach** *pattern* p' *obtained by expending* p *with* a_i **do**
12 compute realizations[w][p'] from realizations[w][p] and realizations[w][a_i];
13 frequency(p') = $\frac{|distinct\ entities\ of\ type\ t\ in\ realizations[w][p']|}{|entities(t)|}$; **if** *frequency*($p'$) $\geq \tau$ **then**
14 patterns[w]=patterns[w] $\cup \{p'\}$;
15 $prune(patterns[w]), realizations[w]$
16 patterns[w]=most_specific_patterns(patterns[w]);
17 **Return**($patterns$)

Our algorithm is initialized with minimal window size (the system default is two weeks) and frequency thresholds (default 0.7), which are iteratively refined: The window size is extended (resp. the threshold it lowered) if no qualified patterns were found, or if the refinement leads to the discovery of additional patterns. The extension granularity (resp. frequency bound reduction) may be determined by the user. Otherwise, the default refinement policy is to alternate between multiplying the window size by two (retaining the threshold as it) and reducing the frequency thresholds by 20% (retaining the window size). This is repeated as long as the refinement leads to new patterns, up to a maximal window size of one year, and a minimum threshold value of 0.2 (All experiments were run with this setting). We chose the above heuristic by examining several alternatives, as elaborated in Section 6, and chose the one with the lowest running time among all heuristics that performed best in terms of $F1$ score evaluations.

We now present the full algorithm, depicted in Algorithm 2. As mentioned above, given an entity type t, our initial entity set S contains all entities of the input type. Users not familiar with the type hierarchy may provide a seed entity e and the system will use $type(e)$ as an input (lines 1-3). To derive $type(e)$ we use an alignment from Wikipedia entities to DBPedia [1]. Then to find all entities of type t we employ a corresponding inverse index.

We first split the timeline into consecutive time frames of size W_{min} (line 7). Next we call (possibly in parallel) the procedure $Mine\_connected\_patterns$, described in Algorithm 1 in Section 4, for all windows (line 9). We iteratively refine the considered windows width (W_{min}) and frequency threshold (τ) (following the heuristics described above), and until a stable result is obtained (lines 10-11). Finally, for each discovered pattern p in window w, its relative frequent patterns are mined as well (lines 14).

Algorithm 2: Find windows and patterns

Input: Wikipedia type t or seed entity e, min. window width W_{min}, frequency threshold τ, relative threshold τ_{rel}
Output: (relative) patterns and their time frames

1 **if** *t is not given* **then**
2 t = type(e);
3 S = get_entities(t);
4 patterns = [];
5 rel_patterns = [];
6 **Frequent patterns Stage;**
7 split the timeline into a set W of consecutive time frames of size W_{min};
8 **foreach** w *in* W **do**
9 patterns[w]= Mine_connected_patterns($S, t, w, \tau, \tau_{rel}$)
10 **if** *patterns==[] or refine?*(W_{min}, τ,*patterns*)==*True* **then**
11 go to line 7 with the updated W_{min}, τ;
12 **Relative frequent patterns Stage;**
13 **foreach** $w \in W$ **do**
14 rel_patterns[w] = Mine_rel_connected_patterns (patterns[w], rel_patterns[w], abstract_actions[w], realizations[w],τ_{rel});
15 **Return**($patterns, rel\_patterns$)

5 USING WINDOWS AND PATTERNS

We employ the discovered windows and patterns to clean and correct Wikipedia entries, as well as to assist users in editing.

Cleaning. An immediate application of the discovered patterns is to alert Wikipedia editors on partial edits from past windows. For that, we examine the discovered windows and identify for each window and pattern (using an efficient outer-join based algorithm, described below, parallelly processed) partial sets of actions that may be extended to a full pattern occurrence. To assist the editor in determining how (if) the partial edit should be completed (or reversed), we present examples of other full patterns.

To explain how the algorithm works, recall from Section 4.2 that, to discover patterns, we iteratively expand the pattern's graph, joining corresponding action relations to form a relation table that captures the pattern realizations. In each such join, the left-hand side (LHS) relation represents the realizations of a (partially growing) portion of the pattern, and the right-hand side (RHS) relation contains the realizations of the added edge. The join conditions assert the (in)equalities of the corresponding graph nodes. To identify *partial updates*, that haven't been properly completed, we similarly traverse the graph. But instead of the abovementioned *join* operator, we employ a *full outer-join* [6], with analogous (in)equality conditions. Note that, unlike the join, the full outer-join also records in the output relation those LHS (resp. RHS) tuples not matching any RHS (LHS) tuple, *padding the missing attribute values with nulls*. In terms of our patterns, partial pattern realizations (resp. action realizations) that are missing a corresponding action (partial pattern) are also recorded in the relation, padded by null values. The incomplete edits can then be easily identified via a selection query retrieving tuples with null values. A result table keeping the attributes of original action relations is kept to record which missing updates cause null values.

Our algorithm for identifying partial updates is depicted in Algorithm 3. For a time window w and a pattern p, it focuses on the entity types in p. It invokes $reduced\_and\_abstract\_actions$ (described earlier), to examine their revision histories and construct the realization relations of their corresponding abstract actions (lines 1–2.) Next we traverse the pattern's graph g_p, and iteratively outer-join the corresponding relations (lines 8-9). We use $p_1 \ldots p_n$ to denote the incrementally growing sub-patterns (from the first singleton edge a_1, to the full pattern p). The array

Algorithm 3: Identifying partial updates

Input: window w, pattern p
Output: partial realizations of p in w

1 let S be the set of entity types in p;
2 call reduced_and_abstract_actions(S, w) to create abstract_actions[w] and realizations[w];
3 let $e_1, \ldots, e_n$ be the edges in the pattern's graph g_p, in some traversal order;
4 let $a_1 \ldots, a_n$ be the corresponding actions in p;
5 $p_1 = \{a_1\}$;
6 $all\_realizations[p_1] = realizations[w][a_1]$;
7 **for** $i = 2 \ldots n$ **do**
8 $\quad p_i = p_{i-1} \cup \{a_i\}$;
9 $\quad$ compute $all\_realizations[p_i]$ from $all\_realizations[p_{i-1}]$ and $realizations[w][a_i]$ using full outer-join;
10 $partial\_r = \{r \in all\_realizations[p] \mid r$ includes a null value $\}$;
11 **Return**$(partial\_r)$

$all\_realizations[p_i]$ is used record the intermediate (possibly incomplete) pattern instantiations. Finally we return all tuples that include null values (lines 10-11). An example of the algorithm execution appears in our technical report [3].

Edit assistance. Update patterns often appear periodically in multiple windows. For example, transfer windows occur each summer with a similar edit pattern. Our system automatically identifies such periodic patterns/windows and provides online edit assistance (via a plug-in) to users that update pattern entities within a given window, suggesting potential update completions, as explained above. The algorithm for identifying patterns that need completion follows similar lines, with the user alerted on partial edits that involve entities that she is updating.

6 EXPERIMENTS

We open this section by describing the experimental setup, the examined datasets, baselines, and evaluation methods. We then present the results, both in terms of running time and quality. Finally, we present a comparative analysis of heuristics, demonstrating the superior performance of the heuristic used by WC.

6.1 Experimental Setup

We have implemented WC as a web browser extension, with backend in Python, frontend in JavaScript, and SQL over pandas as the underlying query engine. All experiments were executed on an Intel i7 2.4Ghz with 96GB RAM and 16 cores server. We ran experiments over Wikipedia datasets and examined the system performance in terms of running times, the quality of the discovered patterns, and the number of detected errors, w.r.t. these patterns.

For the quality experiments (and measuring the running time) we use the default settings of WC. Recall that our algorithm is initialized with minimal window size (default is two weeks) and frequency thresholds (default 0.8), which are refined throughout the computation. As mentioned, the default refinement policy alternates between multiplying the window size by two and reducing the frequency thresholds by 20%, up to at most one year window and a minimal 0.2 frequency. For other experiments, that test the effect of each parameter, we vary the given parameter while setting all others to default values, as explained below.

Settings. To demonstrate the operation of WC in different entity domains, we examine here three Wikipedia domains: soccer (including players, teams, leagues, etc), cinematography (actors, movies, awards, etc) and US politicians (specifically US senators). To derive patterns (and correspondingly identify potential edit errors) we used the revision history for the year 2018. We then validated the signaled potential errors w.r.t. edits recorded in the revision history of 2019. To further assess (resp. validate) the identified patterns (signaled errors), we have also consulted three domain experts - one expert per each of the three domains.

For the soccer domain, we used major European leagues' soccer players for our seed set of entities. For the cinematography domain, we used actors from Hollywood-produced movies for the seed set. Lastly, in the politicians' domain, we used US senators for the seed entity set. In each domain, we considered different sizes of seed sets by randomly choosing between 100-1K entities from the respective seed type. We run each experiment 5 times and show the average running time (the variance was below 5%). For the entities selection, we used the "recently edited" criterion (edited in the last year of 2018) to focus on active pages with edits that may contribute to the mining process, and may also contain errors. Following Algorithm 2, we also considered related entity types and extracted their revision history in the corresponding period.

Due to the lack of an appropriate API, obtaining the Wikipedia data required crawling and parsing entities and it's revision logs. Nevertheless, we gathered data for 100K entities - about 10^{th} of the million frequently edited Wikipedia's entities [4].

Algorithms. The core of WC is Algorithm 2 (referred in the sequel as WC) which identifies time windows of interest and corresponding edit patterns. A main ingredient of WC is the pattern mining procedure depicted in Algorithm 1 (referred in the sequel as PM) that given a specific window w and frequency threshold τ, identifies the most specific frequent patterns in w (w.r.t. the seed type of interest). As explained in Section 4, PM refines conventional graph mining algorithms [15] by introducing two dedicated optimizations: (1) an efficient join-based SQL computation of patterns realization and frequencies, and (2) an incremental computation that avoids a full materialization of the edits graph. To demonstrate the importance of these two optimizations, we examine the running times of the following four algorithm variants.

- PM, our mining algorithm.
- PM^{-join}, a restricted variant of PM without our dedicated join-based queries. Instead, pattern realizations and frequencies are computed via conventional main memory nested loop.
- PM^{-inc}, a restricted variant of PM that does not utilize our incremental, on-demand graph construction. Instead, the full edits graph for the given window is materialized then given as input to the mining process (but patterns realization/frequency is still computed via our join-based queries).
- $\text{PM}^{-inc,-join}$, conventional graph mining without our two optimizations. The edits graph for the window is materialized as input to the mining process, with the pattern realizations/frequencies computed via the main memory nested loop.

Note that direct comparison to leading graph mining baselines is not possible due to their use of different frequency metric (not capturing connectivity property and relativity to a specific type) and lack of support for type hierarchy. We have thus adapted the most relevant variant to our context, denoted by $PM^{-inc-join}$, and benchmark w.r.t. it. See discussion in Section 2.

6.2 Running Time Analysis

Next, we examine how the running time is affected by (1) the size (number of entities) of the seed type of interest, (2) the frequency threshold, and (3) the window size. In each experiment, we vary one parameter while setting the others to a default value (500 seed entities, 0.7 frequency, and two weeks, resp.). As the results for the different domains show similar trends, we present here a representative set of experiments for the soccer domain.

Note that, as is common in graph mining algorithms, PM^{-inc} and PM$^{-inc,-join}$ require the full edits graph for the given window to be materialized. However, materializing this graph, even for relatively small time windows, can be infeasible. Indeed, our experiments show that even when considering a two-week time window, only 100 seed entities, and revision histories only of entities reachable from the seed set, the graph construction exceeded 24 hours (the time limit for the graph materialization). This is due to the dense connectivity of the Wikipedia graph[6] [4], the previously mentioned high volume of edits, and the lack of adequate API, as noted above. Thus, as the graph must be constructed for each considered window, we initially focus only on the two feasible algorithms: PM and PM^{-join}, with the infeasible algorithms evaluated over reduced inputs. We report below the sizes of the partial graphs built by PM and PM^{-join}. For intuition on the relative savings, we note that the graph for the 100 seed entities during these 2-weeks contains over 100K entities.

Seed set cardinality. We start by examining the running time as a function of the size of the seed set. Naturally, the more entities in the seed set, the more related updates need to be examined and the more revision histories are processed. Consequently the running time of both PM and PM^{-join} increases, as illustrated in Figure 4(a). The threshold is set here to the default value of 0.8 and the window is the month of August. Similar results are obtained for other thresholds/months. Next to the size of each seed set, we give (in parenthesis) the overall number of related entities (graph nodes) processed by the algorithm. In each column, the upper part shows how much time (in hours) it took to parse the revision history of the relevant entities and extract the reduced updates set. This is naturally identical for both algorithms (as they only differ in the computation of pattern realizations/frequencies). It should be noted that this time would be much shorter if Wikipedia had provided a more convincing API for its revision logs or, publicly-available structured revisions database. The lower part of each column shows the running time dedicated to the pattern mining itself. We can see that is significantly shorter for PM that employs our efficient join-based queries. For PM the pattern mining time only marginally grows when the seed set size increases and stays below 15 min, which is very reasonable for offline computation.

Frequency threshold. Next, we examine the running time as a function of the frequency threshold. The seed set size is set to a default size of 500 and the window is the month of August. (Similar results are obtained for other sizes/months). The lower the threshold, the more potential patterns (and revision histories of involved entity types) need to be examined, and, consequently, the processing time of both algorithm increases, as illustrated in Figure 4(b). The processing time for the revision logs is the same in both algorithms, but PM mines the patterns much faster. Again, for PM the pattern mining time increases only moderately when the threshold decreases and stays below 15 min.

Window size. In this experiment, we measure the preprocessing time for varying window sizes. Figure 4(c) illustrates the processing time for 2, 4 and 8 weeks window. Specifically, we see here the running times for the first two weeks of August, the whole month of August, and the two months July and August, but similar results are obtained for other similar-length windows. (The seed set size here is again set to default size of 500 and the frequency default 0.8. Similar results are obtained for other sizes/frequencies). Naturally, the larger the window, the more updates need to be processed and as a result, more patterns may occur. Consequently, the running time increases. Again, the processing time for the revision logs in both algorithms is the same, but PM mines the patterns much faster.

Parallelism. So far we examined the performance of the PM component of WC. To complete the discussion we now examine the full operation of WC, highlighting, in particular, its embarrassingly parallelized nature. Recall that WC splits the timeline into non-overlapping windows that may be processed in parallel. Similarly, independent entity types can be processed in parallel. This is easily exploitable in a multi-core setting as shown in 4(d). We focus here on the pattern mining process (the revision logs processing shows similar trends). The figure shows the time in minutes (in log scale) of the pattern mining computation for a single core vs 16 cores, for varying sizes of seed entity sets. As before, next to the size of each seed set we give in parenthesis the overall number of related entities (nodes) processed by the algorithm. Note that the numbers here are the *total number of nodes* processed through all iterations, for all examined windows and threshold values. Running on 1000 entities takes less than one minute on a single core. 5K entities need 6 minutes to process on one core and about 1 minute on 16 cores. For 100K entities - the largest entities set generated in the algorithm execution on the three domains mentioned above - it took 58 minutes on one core and about 15 minutes on 16 cores. Overall, the parallelization speedup is about 4x.

Based on known statistics of approximately 5.9 million Wikipedia entities (one million of them are of mid-to-high importance) [4], given a preprocessed Wikipedia revisions database/graph (which unfortunately is currently not publicly available), running on all Wikipedia entities will take about six hours (one hour on mid-to-high importance entities) on a 16 core server.

Experiments with small data. As mentioned above, the materialization of the entire edits graph of Wikipedia, which is a necessary input for PM^{-inc} and $PM^{-inc-join}$, takes impractical time. To, nevertheless, evaluate the efficiency of these two algorithms, we also conducted experiments over considerably smaller subsets of the Wikipedia graph. Over such small instances, the running time is less meaningful, however, we can focus instead on the number of considered pattern candidates as an indication of the efficiency of these algorithms. Note that, since this experiment is only possible over small data, typically negligible amounts of noise become significant, and since the number of seed entities is small, many of the identified candidates will exceed the threshold. Therefore, we do not examine any quality indicators.

Concretely, we examined a small subset of Wikipedia, consisting of 10 seed entities from the soccer domain, and all the revisions of these entities that occurred within an arbitrarily chosen two-week period. We constructed the corresponding edits graph, containing the seed entities and a close (2-reachable) neighborhood of these seeds in two phases, as follows. We first added to this graph all the entities that are connected within one link from the seeds and were also edited in the chosen time window, and then we also added, analogously, another layer of neighboring entities - all the entities that are connected within one link to the previously added entities, and were also edited in the chosen time window. We did not extend the graph further, as it could not be materialized within the time frame we defined. The above construction resulted in a graph with roughly 10K entities.

We compared the performance of PM^{-inc} and $PM^{-inc-join}$ over this graph, to that ofPM (our pattern mining algorithm) and PM^{-join} (which does not include our dedicated join-based

[6]Wikipedia contains about 6 million entities (of which 4 million are considered of marginal importance) and over 80 million internal links as to 2010.

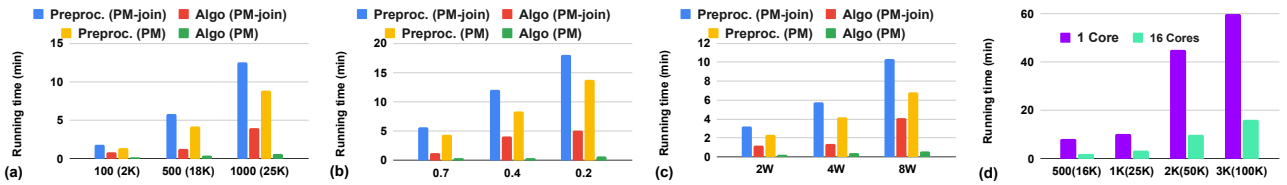


Figure 4: Running time when varying the (a) DB size (b) threshold (c) window size (d) WC execution time on 1 vs 16 cores

queries) over a Wikipedia subgraph of the same size. Recall thatPM and PM^{-join}, in contrast to PM^{-inc} and $PM^{-inc-join}$, do not receive the complete graph as input, rather create the relevant edits subgraph (of the Wikipedia graph) incrementally on-the-fly. Therefore, to ensure a meaningful comparison, we used as input a set of 200 seeds, as this results in subgraphs of roughly 10K entities (which is also the size of the input graphs for PM^{-inc} and $PM^{-inc-join}$). Moreover, as we focus solely on the number of considered candidates, this value will be the same for all variants ofPM, when employed over the same graph, as the frequency definition is identical for all baselines. Therefore, the result will be the same forPM and PM^{-join}, and also the same for PM^{-inc} and $PM^{-inc-join}$. Hence, we essentially compare only two approaches in this experiment (receiving the complete graph in advance versus computing a more relevant subgraph on-the-fly).

The results show that PM^{-inc} and $PM^{-inc-join}$ consider more candidates (524), compared toPM and PM^{-join} (125). This demonstrates the superiority of our incremental graph construction approach, which prunes many of the irrelevant candidates.

6.3 Quality Analysis

To assess the usefulness of WC for error detection we evaluated the quality of the discovered patterns and the validity of the potential errors signaled using these patterns. We employed WC, over the subsets of the Wikipedia 2018 revision log relating to the domains of *soccer*, *cinematography*, and *US politicians*, with the corresponding seed sets consisting of 1000 entities.

Ground truth patterns. To evaluate the correctness and coverage of the detected patterns, we asked each of the three experts to provide a comprehensive list of common periodic update patterns, in structured data. The soccer expert provided 11 such patterns (e.g., the page of a player that won the "Goal of the Month" award should link to the page of the award and vice versa). The cinematography expert provided 8 patterns (e.g., a TV series page should point to all the pages of its specific seasons). Lastly, the politics expert provided 5 patterns (e.g., the page of a newly-elected senator points to her predecessor's page and vice versa).

Discovered patterns and detected errors. Interestingly, the patterns derived by WC are a proper subset of the set of patterns provided by the experts, implying 100% precision. In terms of recall, our algorithm detected 9 (out of the 11) soccer-related patterns, 7 (out of the 8) cinematography related patterns and 4 (out of the 5) US politicians related patterns, yielding an average recall of 83.3% across all the domains. The discovered patterns were then used by WC to detect erroneous updates.

Running Algorithm 3 on the 2018 revision log we have identified 3743 potential errors for the soccer domain, 2554 potential errors for the cinema domain and 1125 potential errors for US politicians. To determine which of these are actual errors, we ran a two-step verification process. First, for each signaled potential error (partial pattern occurrence) we examined whether it still existed after the 2019 updates had been applied. Errors that were eliminated (corrected) are considered true errors. Note, however, that the remaining set may still include actual errors that went unnoticed. To determine how many such signaled, unnoticed errors Wikipedia still contains, we sampled 50 such errors per pattern and asked the relevant domain expert to determine their validity. Next, examples and results of discovered patterns are provided.

Soccer. Out of the 3743 signaled potential errors, 2680 were corrected in 2019 (71.6%). From the remaining examined cases, 82.1% were indeed verified as actual previously unnoticed errors.

To illustrate, the simplest pattern detected in the soccer domain indicates that, after joining a new club, the page of the player should link from the *career* table to the page of the club, which, in turn, should add a link to the player's page in the *current squad* table. This pattern has a frequency of 0.8 in the window consisting of the first week of August. Out of the 50 sampled errors for this pattern (partial pattern occurrences), 48 indeed turned out to be previously unnoticed errors (96%). A more complex pattern includes also the deletion in the player's page of the link to the old club, and vice versa. This pattern has a lower frequency (0.4) and a wider window size (the first two weeks of August). Here, out of 50 sampled potential errors, 44 were verified as actual errors (88%). An example of such an error, detected by the algorithm, relates to the page of Nikola Mitrovic, a player that switched leagues. His new club, ZTE, added him to its current squad table, while the previous club, Kesla, did not remove him. Similarly, Aleksandrs Cauna's page was updated when he joined his new club Jelgava, whereas the page of RFS, his old club, still pointed to his page past the transfer window. A relative frequent pattern, that the algorithm detected, includes an update of the *current league* link in the player's page. While this pattern is much less frequent (since a player may move to a club in the same league, in contrast to the previously mentioned patterns, where a violation almost certainly results in "incomplete" data), its relative frequency, nevertheless, exceeds the threshold. Out of the 50 detected potential errors, 14 were indeed actual previously unnoticed errors.

Cinematography. One example of a detected pattern relates to an actor/actress winning the Oscar award: the page of the winner should link to the page of the award and vice versa. In terms of quality evaluation, out of the 2554 signaled potential errors, 1731 were corrected in 2019 (67.8%). Of the remaining cases, 81.2% were determined to be true unnoticed errors.

US Politicians. An illustrative example of a discovered pattern in the US politics domain pertains to the election of a new senator. Given such an event, the pages of the new senator and the relevant state must point to each other, and also a link to the page of the previous senator is removed from the page of the state. The page of the previous senator should still point to the state, since the only modification relates to the adjacent text, detailing the period during which she held office. Out of the 1125 signaled potential errors, 728 were corrected in 2019 (67.8%). Of the remaining cases, 78.1% were determined to be previously unnoticed errors.

Insights. To conclude, we discuss insights derived from the above evaluation, that reaffirm the distinction between our intended use-cases and those addressed by previous works. As mentioned in the Introduction, our solution focuses on patterns that are associated with a well-defined time window, complementing existing solutions that target 'window-less' constraints. Indeed, for all the discovered patterns, a statistically significant time window was identified. In contrast, of the few overlooked patterns,

Table 1: Sample of heuristics test

(w, τ)	Running time (min)	Precision	Recall	F1 Score
2.0x, 20%	**2**	**1**	**0.84**	**0.91**
1.0x, 20%	1.2	0.88	0.68	0.77
2.0x, 0%	1.2	1	0.75	0.86
1.5x, 10%	3.2	1	0.68	0.81
3.0x, 40%	1.5	0.75	0.88	0.81

two are not clearly associated with any time window. This further reinforces the contrast between our solution and other works.

6.4 Parameter Tuning

When refining the two parameters across different iterations, *PM* alternates between multiplying the window size by two and reducing the frequency threshold by 20%. To arrive at these values, we performed a grid search, selecting the parameters that led to the fastest running time among the options that yield the best $F1$ score (w.r.t. patterns provided by experts). We checked combinations of reducing the threshold by X and multiplying the window size by Y, where X ranges from 1% to 100%, in steps of 5%, and Y ranges from 1.5 to 5, in steps of 0.5. In terms of the bounds for the above parameters, the window size is restricted to the range of two weeks to one year, while the threshold is restricted to $[0.2, 0.7]$. These intervals were also derived via an analogous grid search over various ranges. A sample of the results is depicted in Table 1, where the left column provides the combination of the changes in the values of the window size and the threshold. Note that the first row pertains to the combination used by `WC`.

These results demonstrate the advantages of our balanced approach, compared to more extreme approaches. Namely, opting for very small changes to the parameters increases the running time and lowers the recall. The recall drops because WC would terminate at an early stage, as new patterns are not likely to be discovered compared to the previous iteration. At the other extreme, drastically changing the parameter values, while improving the running time, lowers the precision score. The latter effect is due to quickly reaching iterations where the time window is large and the threshold is low, causing WC to discover erroneous patterns, whereas WC with our heuristic would terminate prior to this point.

7 CONCLUSION

This paper presents WC, a Wikipedia plug-in assisting editors in maintaining the correctness of inter-links. Given an entity type of interest, our efficient, highly parallelizable algorithm identifies relevant edit patterns across revision histories of entities of related types, along with time windows in which partial edits are acceptable. The discovered patterns/windows are then used by WC to alert editors on past edits that appear incomplete, and provide users with on-line assistance as they update Wikipedia. Our experiments with Wikipedia data demonstrate the efficiency and effectiveness of our approach in identifying and correcting errors.

There are several directions for future research. As our work considers inconsistencies in structured parts of Wikipedia, expanding our approach to consider free text, in particular parts related to the inter-links, is a challenge. Another intriguing future direction is enriching the expressiveness of the patterns to support value-specific instantiations (e.g., a pattern specific to PSG, but not to football clubs in general). Finally, applying our ideas to other domains where revision histories are available and link consistency is important (e.g., software repositories) is another challenge.

Acknowledgements This work has been partially funded by the Israel Science Foundation, the Binational US-Israel Science Foundation, and the Tel Aviv University Data Science center.

REFERENCES

[1] DBPedia. https://wiki.dbpedia.org/.
[2] Time Magazine, "Jimmy Wales". http://content.time.com/time/specials/packages/article/0,28804,1975813_1975844_1976488,00.html.
[3] WiClean Technical Report. http://slavanov.com/research/wc-tr.pdf.
[4] Wiki statistics. https://en.wikipedia.org/wiki/Wikipedia:Statistics.
[5] Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker. Temporal rules discovery for web data cleaning. *PVLDB*, 9(4):336–347, 2015.
[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[7] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.
[8] E. Alfonseca, G. Garrido, J.-Y. Delort, and A. Peñas. Whad: Wikipedia historical attributes data. *Language resources and evaluation*, 47(4), 2013.
[9] A. Assadi, T. Milo, and S. Novgorodov. Cleaning data with constraints and experts. In *WebDB*, pages 1:1–1:6, 2018.
[10] M. Atzori, S. Gao, G. M. Mazzeo, and C. Zaniolo. Answering end-user questions, queries and searches on wikipedia and its history. *IEEE Data Eng. Bull.*, 39(3):85–96, 2016.
[11] M. Bergman, T. Milo, S. Novgorodov, and W. C. Tan. Query-oriented data cleaning with oracles. In *SIGMOD*, pages 1199–1214, 2015.
[12] T. Bleifuß, L. Bornemann, T. Johnson, D. V. Kalashnikov, F. Naumann, and D. Srivastava. Exploring change: a new dimension of data analytics. *Proceedings of the VLDB Endowment*, 12(2):85–98, 2018.
[13] S. Bostandjiev, J. O'Donovan, C. Hall, B. Gretarsson, and T. Hollerer. Wigipedia: A tool for improving structured data in wikipedia. In *2011 IEEE Fifth International Conference on Semantic Computing*, pages 328–335, Sep. 2011.
[14] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *Proc. VLDB Endow.*, 6(13):1498–1509, Aug. 2013.
[15] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7), 2014.
[16] W. Fan, X. Wang, Y. Wu, and J. Xu. Association rules with graph patterns. *Proceedings of the VLDB Endowment*, 8(12):1502–1513, 2015.
[17] B. Fetahu, A. Anand, and A. Anand. How much is wikipedia lagging behind news? *CoRR*, abs/1703.10345, 2017.
[18] J. Ge and Y. Xia. Distributed sequential pattern mining in large scale uncertain databases. In *PAKDD*, pages 17–29. Springer, 2016.
[19] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9):625–636, 2013.
[20] S. Goldberg, T. Milo, S. Novgorodov, and K. Razmadze. WiClean: a system for fixing Wikipedia interlinks using revision history patterns. *PVLDB*, 12(12):1846–1849, 2019.
[21] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *ACM sigmod record*, 29(2):1–12, 2000.
[22] A. Heidari, J. McGrath, I. F. Ilyas, and T. Rekatsinas. Holodetect: Few-shot learning for error detection. *arXiv preprint arXiv:1904.02285*, 2019.
[23] S. Heindorf, M. Potthast, B. Stein, and G. Engels. Towards vandalism detection in knowledge bases: Corpus construction and analysis. In *SIGIR*, 2015.
[24] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(1):75–105, 2013.
[25] P. Kin-Fong Fong and R. Biuk-Aghai. What did they do? deriving high-level edit histories in wikis. 01 2010.
[26] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery*, 11(3):243–271, 2005.
[27] J. Lajus, L. Galárraga, and F. Suchanek. Fast and exact rule mining with amie 3. In *European Semantic Web Conference*, pages 36–52. Springer, 2020.
[28] N.-T. Le, B. Vo, L. B. Nguyen, H. Fujita, and B. Le. Mining weighted subgraphs in a single large graph. *Information Sciences*, 514:149–165, 2020.
[29] M. Muzammal and R. Raman. Mining sequential patterns from probabilistic databases. *Knowledge and Information Systems*, 44(2):325–358, 2015.
[30] S. Ortona, V. V. Meduri, and P. Papotti. Rudik: Rule discovery in knowledge bases. *Proceedings of the VLDB Endowment*, 11(12):1946–1949, 2018.
[31] D. Raghu, S. Nair, and Mausam. Inferring temporal knowledge for near-periodic recurrent events. 2018.
[32] N. Rajkumar, M. Karthik, and S. Sivanandam. Fast algorithm for mining multilevel association rules. In *Proc. of TENCON*, pages 688–692, 2003.
[33] A. Sarabadani, A. Halfaker, and D. Taraborelli. Building automated vandalism detection tools for wikidata. WWW 2017 Companion.
[34] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of EDBT*, pages 1–17, 1996.
[35] O. Sunercan and A. Birturk. Wikipedia missing link discovery: A comparative study. In *2010 AAAI Spring Symposium Series*, 2010.
[36] T. P. Tanon, C. Bourgaux, and F. M. Suchanek. Learning How to Correct a Knowledge Base from the Edit History. In *WWW*, 2019.
[37] T. Tran and T. N. Nguyen. Hedera: Scalable indexing and exploring entities in wikipedia revision history. pages 297–300, 2014.
[38] R. West, A. Paranjape, and J. Leskovec. Mining missing hyperlinks from human navigation traces: A case study of wikipedia. In *WWW*, 2015.
[39] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *Proc. of KDD*, pages 286–295, 2003.
[40] R. Ying, A. Wang, J. You, and J. Leskovec. Frequent subgraph mining by walking in order embedding space. 2020.
[41] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1-2):31–60, 2001.

Automating Data Quality Validation for Dynamic Data Ingestion

Sergey Redyuk, Zoi Kaoudi, Volker Markl
Technische Universität Berlin
[sergey.redyuk,zoi.kaoudi,volker.markl]@tu-berlin.de

Sebastian Schelter
University of Amsterdam
s.schelter@uva.nl

ABSTRACT

Data quality validation is a crucial step in modern data-driven applications. Errors in the data lead to unexpected behavior of production pipelines and downstream services, such as deployed ML models or search engines. Typically, unforeseen data quality issues are handled via manual and tedious debugging processes in a reactive manner. The problem becomes more challenging in scenarios where large growing datasets have to be periodically ingested into non-relational stores such as data lakes. This is even worse when the characteristics of the data change over time, and domain expertise to define data quality constraints is lacking.

We propose a data-centric approach to automate data quality validation in such scenarios. In contrast to existing solutions, our approach does not require domain experts to define rules and constraints or provide labeled examples, and self-adapts to temporal changes in the data characteristics. We compute a set of descriptive statistics of new data batches to ingest, and use a machine learning-based novelty detection method to monitor data quality and identify deviations from commonly observed data characteristics. We evaluate our approach against several baselines on five real-world datasets, on both real and synthetically generated errors. We show that our approach detects unspecified errors in many cases, outperforms other automated solutions in terms of predictive performance, and reaches the quality of baselines that are hand-tuned using domain expertise.

1 INTRODUCTION

Data-driven decision making is becoming the norm in modern enterprises and organizations, and requires maintaining and regularly updating large datasets, often collected in non-relational stores such as data lakes. A critical step in these scenarios is data quality validation, as the quality of the derived insights and decisions crucially depends on the quality of the collected data [42]. Incorrect or missing data can lead to wrong business decisions and problems in downstream data consumers, such as machine learning (ML) models or search engines [1, 17, 43], and even crash systems, e.g., due to null-pointers originating from missing data. Common sources of errors are bugs in external data sources and data preprocessing code (e.g., when a data engineer accidentally changes a time measurement from seconds to milliseconds in a data-producing pipeline). Such errors often corrupt large parts of the data to ingest and can immediately lead to devastating consequences, e.g., wrong predictions of ML models that consume the data [37]. In this work, we focus on automating the detection of such data quality issues.

We address the following real-world example scenario. Consider a data engineering team at a retail company maintains a search engine for products. To keep the search engine up-to-date, it deploys a pipeline that regularly ingests and indexes external product data from various heterogeneous sources, such as web crawls, log files, key-value stores, or upstream data pipelines. If a data source introduces errors in the data to ingest, such as missing values or wrong encoding of strings, then the products will not be indexed correctly or, even worse, cause the ingestion process to crash. Such data issues are typically handled reactively: the engineering team discovers data issues via alerts from devops engineers, bug reports, and customer reviews. The data is then manually fixed and back-filled. Handwritten code is added to the data pipeline in retrospect to catch the observed type of errors in the future [43].

In this paper, we propose an approach to automate data quality validation in scenarios where large partitions of a growing dataset have to be regularly ingested into a common data store such as a data lake. While relational databases enforce a schema and integrity constraints for their data [13], many modern applications rely on non-relational data stores. Pipelines that do not specify a particular schema or constraints on the data are often much cheaper to operate in cloud environments (e.g., using S3 as a distributed filesystem for storing the data partitions and Apache Spark for processing them).

In contrast to existing work on fine-grained error detection [1, 17, 27, 29, 36, 47], we focus on scenarios where systems regularly ingest batches of external data, and data errors corrupt a large fraction of the batch [42] (Section 3). In the aforementioned retail example, a few missing product reviews in a partition might not cause issues in the downstream systems, as they are programmed to handle that (e.g., by using missing value imputation strategies). However, an unusually high fraction of missing values in the review description is an indicator of a severe problem in one of the external data sources.

We automate the detection of six types of errors (explicit and implicit missing values, numeric anomalies, typos, swapped fields for numeric and textual attributes) as follows: we leverage previously ingested data batches as "positive" examples of "acceptable" data and use a machine learning approach to identify new batches that significantly deviate from the previously observed data. Specifically, we compute a set of descriptive statistics over the ingested data and train a novelty detection ML model [30, 31] to learn the characteristics of the "acceptable" data. We apply the ML model on new data batches to ingest, in order to identify potentially erroneous data batches that significantly differ from previously observed data (Section 4).

Our approach provides several advantages over existing work. First, it does not require domain experts to design and maintain large numbers of rules [3, 20, 43]. Devising such rules and constraints is a very tedious and expensive process as the datasets found in enterprises are typically large and messy [45], especially if they originate from the integration of different external data sources. Secondly, our approach is computationally efficient as the descriptive statistics we apply can be computed in a single

pass over the data. Our novelty detection model has a low number of parameters to optimize. Finally, our automated approach performs well in cases where the data characteristics change over time, in contrast to rule- and constraint-based approaches [20, 43] that require a manual redefinition of rules and constraints.

We evaluate our approach by comparing its predictive performance to automated and hand-tuned variants of the following state-of-the-art solutions: Tensorflow Data Validation [6], Deequ [43], and statistical testing [32, 41]. Then, we evaluate the sensitivity of our approach towards six types of errors (explicit and implicit missing values, numeric anomalies, typos, swapped fields on numeric and textual attributes) and the predictive performance under various error magnitudes (1, 5, 10, 20, . . . , 80%) in a controlled environment for datasets with synthetically generated errors. Finally, we evaluate the detection quality of our approach over time, as (a) the size of the training set for the novelty detection algorithm grows continuously, and (b) its data characteristics change over time. In summary, we make the following contributions:

- We propose an approach to automate data quality validation for data that is periodically ingested into non-relational stores. In contrast to existing solutions, our approach does not require domain experts to define rules or labeled examples, and self-adapts to temporal changes in the data characteristics (Sections 3 & 4);
- We discuss how to apply our approach efficiently via a novelty-detection ML model trained on data quality metrics of the data (Section 4);
- We evaluate our approach against existing baselines on five real-world datasets with real and synthetically generated errors. We find that our approach detects the unspecified errors in many cases under varying error magnitudes, outperforms other automated solutions in terms of predictive performance, and reaches the ROC AUC score of baselines hand-tuned with domain expertise (Section 5).

2 BACKGROUND

In the context of this work, we understand data quality validation as the process of checking that the input data meet the needs of a data-driven application or its underlying business process, where these specific needs are either formulated explicitly with the data standards and policies or assumed implicitly by the application logic. The concept of data quality is broadly defined as a measure of the fitness of the data to their intended uses and purposes [11]. To identify how well the data fit for the intended purpose, the wast body of knowledge [5] suggests several data quality dimensions, such as data *accuracy* (the degree to which the data correctly represent the real-world entity it models), *completeness* (the degree to which the data contain the necessary attributes to model the entity), *validity* (the degree to which the data are stored or represented in a format that is consistent with the domain of values), and others. In practice, data quality is assessed with a set of quantitative metrics that are associated with the aforementioned data quality dimensions. In this section, we briefly introduce the data quality metrics that we leverage in our approach and the machine learning-related background for novelty detection.

Data quality metrics. We consider several quantitative statistics that can be used to identify data quality issues [18]: (*i*) *completeness* - the ratio of non-missing values to the number of records in the data; (*ii*) *the number of distinct values*; (*iii*) statistics for numeric data types, such as *maximum, mean, minimum, and standard deviation*, (*iv*) the ratio of *occurrence for the most frequent value*, etc. These statistics are commonly used in database engines, for data profiling and data quality validation [18] to summarize data of interest and often act as a proxy for the state of data quality. Furthermore, most of the statistics can be cheaply computed in a single scan over the data, except for the number of distinct values and the ratio of the most frequent value, which are typically approximated with the hyperloglog and the count-min sketches respectively [8, 12].

Novelty Detection. Novelty detection is a machine learning technique that aims to identify new patterns and signals that were not present in the training data [30]. It is closely related to anomaly detection as both techniques look for patterns in data that do not conform to the expected behavior [7]. The difference is that anomaly detection assumes that outliers are already present in the training data. In contrast, novelty detection is designed for cases where we only have access to "positive" examples.

Novelty detection is a form of one-class classification [46] (due to the absence of negative examples). Novelty detection algorithms model the data and check whether previously unseen data points resemble the characteristics of the modeled data (i.e., inliers) or deviate from the expected behavior (i.e., outliers). The decision whether or not a new object (i.e., data point) is an outlier against a set of known objects follows the continuity assumption (i.e., two data points that are close in the feature space represent two objects with the resemblance in real life) and usually focuses on distance measures. Common example algorithms in this area are one-class SVMs [44] and isolation forests [26]. For an in-depth overview of the one-class classification problem and novelty detection algorithms, we recommend the reader to refer to Tax [46] and Chandola et al. [7].

3 PROBLEM STATEMENT

In this section, we introduce the problem and its formal definition.

Overview. We address the problem of automating the validation of data quality on dynamic data without relying on domain expertise (e.g., manually specified rules and labeled erroneous data records). As outlined in the running example, we focus on scenarios where data pipelines regularly ingest large batches of potentially erroneous external data and face errors that corrupt a large fraction of the batch.

State-of-the-art solutions in data quality validation typically require domain knowledge to specify explicit rules, constraints, patterns, or labeled examples to verify data quality [6, 18, 20, 43]. They, however, fall short in several cases: (*i*) incomplete domain knowledge (i.e., when data depict complex processes that even domain experts cannot fully comprehend or when the domain expert is unavailable at the given time), the solutions mentioned above might perform poorly both due to false alarms and missed errors as the specified set of rules or labeled examples are insufficient to capture potential errors; (*ii*) manual monitoring of data pipelines to detect data quality issues or deployment of staging environments for software testing are often too costly or time-consuming, and are only conducted reactively; (*iii*) the characteristics of the data might slowly change over time, which implies that manually specified rules have to be constantly adapted and maintained.

These challenges motivate an automatic approach to data quality validation that does not rely on manually specified rules or labeled examples and self-adapts to changes in data characteristics.

Assumptions. For the given use case of the regular ingestion of large batches of a growing dataset, we consider previously observed and successfully ingested data partitions to be of "acceptable" data quality. This assumption is based on our experience with real-world use cases: It is common in production to define principal business and operational performance indicators and monitor them carefully to evaluate business outcomes. For our retail company running example, products that are placed in the wrong category due to various errors lead to negative customer reports or low service ratings, or via incident reporting and tracking systems. This negative feedback serves as a proxy that affects key performance indicators and catches the attention of the responsible staff at some point in time. This, in turn, triggers retrospective analysis. If devastating errors would have occurred in the previously observed data partitions, they would have been detected and fixed after a given time. If errors do not trigger a negative response from the devops engineers or the business after some period of time, we assume that the downstream task is robust to them. Furthermore, existing error detection or data quality validation methods require domain expertise. We focus on real-world scenarios where domain expertise is not available that, in turn, render the majority of data quality monitoring tools inapplicable.

Formal problem statement. Given a structured dataset D of chronologically ordered partitions $d_1, \ldots, d_{t-1}$, each having domain $A = A_1, \ldots, A_M$, we have to predict upon the arrival of a new partition d_t whether this partition is of acceptable quality or it is potentially corrupted w.r.t. a set of data quality metrics $Q = Q_1, \ldots, Q_G$. We map this problem to a "one-class classification" problem [46] where every partition d_i is represented by a feature vector $\mathbf{x}_{d_i} = (x_1, \ldots, x_G) \in \mathbb{R}^G$ of the data quality metrics Q that are computed on every attribute A_i of that partition and a boolean label y_{d_i}, which denotes whether the quality of the batch is acceptable or not. However, we only have access to positive examples during training (hence the term "one class" classification). The classification task is to decide whether a future batch d_t can be considered of acceptable quality (i.e., represents an inlier) or deviates from the state of data quality of the previously observed data batches (i.e., represents an outlier). The main challenge is to model the "acceptable" data in an automated manner, without external specification of the domain or examples of "erroneous" data that have insufficient data quality.

4 APPROACH

Next, we discuss our approach for automating data quality validation of newly observed data batches based on the problem definition we presented in the previous section.

Overview. Figure 1 illustrates our approach: for every observed partition $d_1, \ldots, d_{t-1}$, we model the features $\boldsymbol{x}_{d_i}$ via a set of descriptive statistics computed from the partition ❶. We train a novelty detection model [38] on the resulting feature vectors that learns the characteristics of "acceptable" data ❷. In order to check a new data batch d_t, we compute its feature vector $\boldsymbol{x}_t$ via the chosen descriptive statistics ❸. Next, we apply the novelty detection model to label the new batch as acceptable or erroneous based on the learned decision boundaries of the model ❹. With every new data partition d_t, we re-train the novelty detection model as the training set grows with t. Our method can be integrated into data pipelines to raise alerts about potential degradation of data quality automatically. Note that our approach does not rely on domain expertise expressed in the form of rules, constraints, or labeled data. Still, it remains valid in cases where the task definition is relaxed (e.g., domain knowledge is partially available or some error types are expected).

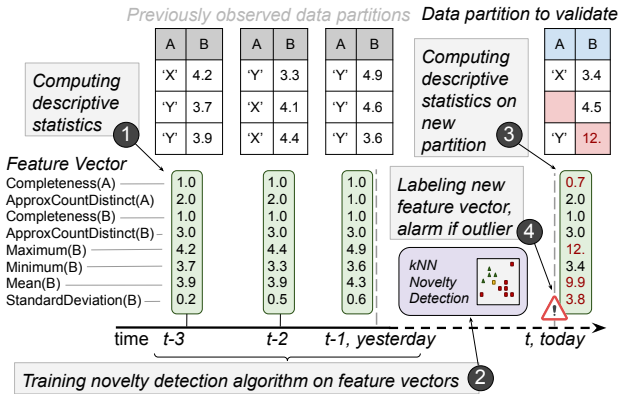


Figure 1: Overview of the approach: for every observed partition (gray tables), we compute a set of descriptive statistics as a feature vector (green, Step 1). We train a novelty detection model that learns the characteristics of acceptable data (Step 2). For the upcoming data partition (blue table), we compute its feature vector (Step 3) and let the model decide whether it is similar to the previously observed data partitions or not (Step 4). In this example, a missing value in column "A" and a numerical outlier in column "B" (red) affect the completeness metric and numeric statistics of the feature vector x_t. That, in turn, raises an alert.

Descriptive statistics as features. For every attribute A_j of the partition d_i, we compute several quantitative measures that correspond to the underlying data quality metrics (see Section 2):

- *Completeness* - the ratio of not-NULL values;
- *Approximate count of distinctive values* - the hyperloglog [12] approximation of the number of distinctive values;
- *Ratio of the most frequent value* - the count sketch [8] approximation of the number of occurrences for the most frequently repeated value, normalized by the batch size;
- *Maximum, mean, minimum, and standard deviation* for numeric data types;
- *Index of peculiarity* [33] for textual data. Index of peculiarity is based on the bi- and trigram tables of a textual attribute and reflects the likelihood of the hypothesis that trigrams in a given word are produced from the same data source that produced the trigram table. This index is originally applied for detection of typographical errors and facilitates detection of typos in text or a "peculiar" occurrence of symbols in words.

$$I(\mathbf{T}) = \frac{1}{2}(\log n(xy) + \log n(yz)) - \log n(xyz) \qquad (1)$$

Equation 1 represents the index of peculiarity for a trigram $\mathbf{T} = (xyz)$, where $n(\ )$ denotes the number of occurrences for a selected bi- or trigram in a textual attribute. Index of peculiarity for a sentence is the root-mean-square aggregation of indices for each trigram that this sentence contains.

Algorithm 1: Pseudocode of our approach.

Input: **t**, query raw data partition; **k**, the number of neighbors;
X, descriptive statistics for previously ingested data partitions;
contamination, the proportion of outliers in X;
dist, distance measure (e.g., Euclidean, Manhattan);
agg, distance aggregation strategy for k nearest data points.
Output: **label**, query data point **t** is inlier/outlier

```
Initialize array statistics; array distances;
        list num_met of metrics for numeric data types;
        list gen_met of metrics for other data types.
foreach attribute A ∈ t do
  metrics = num_met if type(A) is numeric else gen_met
  foreach metric ∈ metrics do statistics.append(metric(A))
end
foreach x ∈ X, tree = BallTree(X, dist) do
  /* .getDist(x, k) returns distances to k nearest
     neighbors of x; agg(array) is an aggregation
     function such as mean, median, or max          */
  distances.append(agg(tree.getDist(x, k))
end
/* percentile(x, q) computes q-th percentile of x   */
threshold = percentile(distances, (1 − contamination))
/* outlier if aggregated distance from t to k nearest
   neighbors exceeds threshold, else inlier         */
return agg(tree.getDist(statistics, k)) < threshold
```

We concatenate attribute-level statistics into a univariate numeric vector. Depending on the number of attributes and their data types, the feature vector varies in length from one dataset to another, where the length remains constant for partitions of the same dataset. We normalize the resulting feature vectors to a scale of 0 to 1. We chose these statistics based on two criteria: (a) low computational complexity and (b) mapping to the error types that often occur in real-world scenarios [47]. For a particular error type that we investigate, we consider statistics that act as proxies for this error type more descriptive than others in detecting data quality degradation. By a proxy we mean a quantitative measure that is expected to change when a particular error occurs (e.g., numeric outliers are likely to affect the statistical distribution of the attribute [18]). There is no single metric that is more descriptive than others for all the given error types. Preliminary results show that specifying only the descriptive statistics that we expect to be changed when an error occurs increases performance of our approach. This happens because, in low-dimensional feature spaces, data points are more distinct and distance-based methods perform better. However, assuming "zero domain knowledge" and unknown error types, we cannot control the choice of descriptive statistics in practice and, thus, train our approach on all statistics. As discussed in Section 2, most of these statistics can be computed in a single scan over the data. Furthermore, we treat the sequence of feature vectors that we collect over time (i.e., $t_{start}, \ldots, t-1$) as separate data points in the training set. Note that this modeling decision does not preserve the order of these feature vectors.

Choice of the novelty detection algorithm. Given the nature of the challenge at hand, i.e., "zero domain knowledge" or unknown error types, only positive examples are available for training. We thus choose one-class classification algorithms (i.e., novelty detection, see Section 2) as the main candidates for our approach. In this work, we considered several candidates for the novelty detection (ND) algorithm: Angle-based Outlier Detector (ABOD), Feature Bagging ensemble for the Local Outlier Factor (FBLOF), Histogram-base Outlier Detection (HBOS), Isolation Forest, and the K Nearest Neighbors algorithm with both the maximum and the mean distance aggregation scheme (KNN and Average KNN, respectively) [30, 31]. To choose one particular ND algorithm for our approach, we conduct preliminary experiments on one dataset (`Amazon Review`, monthly data partition) and three types of errors (explicit and implicit missing values on all attributes, numeric anomalies on the attribute "overall") with 30% of synthetically introduced errors per data batch, in order to determine which algorithm yields better predictive performance on the one-class classification task (for more details, see Section 5). We deliberately chose one dataset and a subset of error types under investigation to avoid overfitting and the selection bias for the evaluation procedure. Table 1 depicts the predictive performance metrics (ROC AUC score [22]) for all the ND candidates, as well as the break-down of the false positive and false negative results. We report the ROC AUC measure as it takes into account both the type-I and type-II errors. Furthermore, it is insensitive to imbalanced datasets and preferred in practice to other performance metrics such as accuracy or F1 score. In our preliminary experiments, we computed other performance metrics alongside the ROC AUC score. We noticed that, since our evaluation scenario introduces a balanced case where a negative counterpart exists for every positive example, accuracy, F1 and ROC AUC scores report similar values. Based on the preliminary results, we chose the k-Nearest Neighbor algorithm with the mean aggregation scheme [38]. This algorithm consistently outperformed other ND candidates on all three error types and produced no false positive results, meaning that no erroneous data batches were labeled as "acceptable". The second best-performing candidate is the Angle-Based Outlier Detection method [23] that yielded comparable predictive performance yet took an order of magnitude longer to train the model and infer the labels.

Nearest-neighbor-based novelty detection. For every data point in the feature space, the k-Nearest Neighbor (kNN) algorithm calculates the average distance to its k nearest neighbors and learns a threshold to decide what data points to consider inliers or outliers [2]. The kNN algorithm has a *contamination* hyperparameter that defines a ratio of data points that are assumed to be incorrectly labeled as inliers. Hence they are labeled as outliers in the training data. This scheme internally translates the one-class classification problem into a standard binary classification problem where the examples of both classes are present. The algorithm utilizes the Ball tree[35] space partitioning data structure - a binary tree where each node represents a multi-dimensional hypersphere (i.e., ball) of partitioned data points. This data structure provides properties that are useful for efficient k-nearest neighbor search. All data points in the training set are represented with distances to their k nearest neighbors. Depending on the design decision, these distances are aggregated into a single numeric value with one of the available aggregation strategies (e.g., mean, median, max). These numeric values are used to learn a decision boundary to differentiate inliers and outliers - a data point is considered an outlier if its aggregated distance to k nearest neighbors exceeds the learned threshold. The threshold is defined with the contamination hyperparameter c that is translated into the $(1-c)$th percentile of the array of aggregated distance for the whole training set. Figure 1 provides a pseudocode representation of the KNN algorithm.

Table 1: Results of the preliminary experiment on performance evaluation for 7 novelty detection algorithms. Three error types under investigation are explicit and implicit missing values, and numeric anomalies, depicted as "Explicit MV", "Implicit MV", and "Anomaly" respectively. We measure predictive performance with the ROC AUC score (AUC), as well as the number of true positive (TP), false positive (FP), false negative (FN), and true negative (TN) results, where FPs are associated with the misclassification rate and FNs - with the false alarm rate.

ND Algorithm	Error type	AUC	TP	FP	FN	TN
One-class SVM	Explicit MV	.9213	178	0	28	150
	Implicit MV	.9213	178	0	28	150
	Anomaly	.9691	178	0	11	167
ABOD	Explicit MV	**.9382**	178	0	22	156
	Implicit MV	**.9382**	178	0	22	156
	Anomaly	.9691	178	0	11	167
FBLOF	Explicit MV	.9353	178	0	23	155
	Implicit MV	**.9382**	178	0	22	156
	Anomaly	.9662	178	0	12	166
HBOS	Explicit MV	.5814	60	118	42	136
	Implicit MV	.5505	60	118	42	136
	Anomaly	.9297	176	2	23	155
Isolation Forest	Explicit MV	.7331	27	151	18	160
	Implicit MV	.5280	27	151	17	161
	Anomaly	.8764	146	32	12	166
KNN	Explicit MV	.9325	178	0	24	154
	Implicit MV	.9325	178	0	24	154
	Anomaly	.9662	178	0	12	166
Average KNN	Explicit MV	**.9382**	178	0	22	156
	Implicit MV	**.9382**	178	0	22	156
	Anomaly	**.9719**	178	0	10	168

Gu et al. [15] present an extensive statistical analysis of nearest neighbor algorithms and report that recent work on this family of methods reaches state-of-the-art performance on novelty detection tasks. Based on the preliminary experiment, we confirm that the kNN novelty detection method performs on par with other approaches or outperformed them, both in terms of the predictive performance and execution time.

Modeling decisions. Next, we discuss several modeling decisions for our kNN-based approach. We choose the Euclidean distance metric as the most commonly used distance measure for the $\mathbb{R}^G$ feature space, and leverage the average distance to k neighbors as an aggregation strategy. Based on preliminary experiments, this decision led to consistently higher predictive performance compared to other settings. Alternative strategies are choosing the largest distance among k neighbors or computing the median. A systematic comparison of kNN algorithms with different distance measures revealed that both the "largest" and the "median" aggregation schemes happen to be less robust than averaging in our setting.

We set the number of neighbors k to aggregate the distance measure to a low factor of five. The variation of this parameter did not lead to significant changes in the predictive performance during the preliminary experiments. The kNN novelty detection algorithm is also parameterized with the contamination parameter [19]. This parameter defines a fraction of data points in the training set to be misclassified as "positive" examples and assumed to be outliers (i.e., false positives). We set the contamination parameter to 1% to keep the ratio of false positives minimal.

Table 2: Characteristics of the datasets. The abbreviations depict, in a direct order, the number of records in the dataset, the number of partitions, the total number of attributes, the average number of records in a data partition, the number of numeric, categorical, and textual attributes. We also report the real-world error types that two datasets with the ground truth, `Flights` and `FBPosts`, contain.

Dataset	Flights	FBPosts	Amazon	Retail	Drug
# records	147640	11157	1494070	541909	161297
# part./attr.	31/9	53/14	1665/9	305/8	3579/6
part. size	~2350	~105	~897	~1776	~45
N/C/T	1/4/0	4/3/2	2/1/4	2/5/1	2/2/1

Dataset	Flights	FBPosts
Errors, %	explicit/implicit missing values, 8-38%	wrong encoding, 16%
	incomplete datetime format, 95%	syntactic errors and translation, 18%
	other syntactic/semantic errors, 60%	

We aim to minimize the number of data points in the training set that are considered to be falsely classified as "inliers". We base this decision on our assumption that all the data partitions are of "acceptable" quality, and no misclassification occurs. Preliminary experiments showed that setting the contamination parameter to 1% leads, on average, to relatively higher predictive performance compared to other values (including 0). Note that automated hyperparameter tuning schemes are challenging in the case of one-class classification problems, as we do not have labels for both of the classes - acceptable and erroneous data.

Application to our example scenario. Based on the running example, imagine the engineering team to apply the proposed approach as a data quality monitoring tool to validate incoming data batches before running data preprocessing and indexing jobs. When a new data batch is examined and no alerts are raised, data pipelines work without any difference and run the downstream preprocessing and indexing job. In case an alert is raised, the team starts a debugging process and applies further error detection and correction strategies. If the method caught the erroneous data batch correctly, the team fixes it and released the quarantined batch back to the pipeline. In the case of false alarms, the data is returned without alterations. The critical point is when the erroneous data batch passes data quality checks and goes further to the downstream pipeline without the errors being fixed (i.e., false positives). In this case, system crashes and degradation in the predictive performance of the underlying ML model might occur.

5 EVALUATION

In this section, we introduce our experimental setup and discuss datasets and metrics for our evaluation. We conduct several experiments. First, we compare the predictive performance of our approach to automated and hand-tuned variants of the following state-of-the-art solutions: Tensorflow Data Validation [6], Deequ [43], and statistical testing [32, 41]. Then, we evaluate the sensitivity of our approach towards six types of errors (explicit and implicit missing values, numeric anomalies, typos, swapped fields on numeric and textual attributes) and the predictive performance under various error magnitudes (1, 5, 10, 20, . . . , 80%)

in a controlled environment for datasets with synthetically generated errors. Finally, we evaluate the detection quality of our approach over time, as (a) the size of the training set for the novelty detection algorithm grows continuously, and (b) its data characteristics change over time.

5.1 Experimental Setup

We evaluate our proposed approach as follows. We experiment with a relational dataset that is partitioned by a chosen temporal attribute (e.g., a creation timestamp for every record). This allows us to simulate our target scenario of the daily ingestion of new data batches in a data pipeline. For every data point that corresponds to a particular day t, we use the previously observed partitions from timestamp 0 to $t-1$ as training data for our approach. Then, we take both the partition d_t and a corrupted version $\hat{d}_t$ as a counterpart, pass it to our model, and have it predict whether the partition is of acceptable data quality or not. Data partitions of acceptable quality are those that do not affect KPIs and usually depend on the downstream ML task. However, to decouple our experimental evaluation from the underlying ML task, we consider partitions of acceptable data quality the ones that do not contain any errors. We apply standard binary classification metrics such as the area under the ROC curve (ROC AUC score [22]) to evaluate how well the approach performs. We also report confusion matrices to analyze misclassification and false alarm rates.

Datasets. We experiment on five publicly available real-world datasets from different application domains. For two of them, we have access to both the erroneous and the cleaned versions of the data [25]. The other three do not contain any errors, we thus generate the errors synthetically [9, 14, 16]. For details, see Table 2.

Datasets with ground-truth errors. The `Flights`[1] dataset [25] contains flight status data that is aggregated from 38 different data sources (the airline and the airport websites, third-party web resources). Each record represents a particular flight on a particular day and includes attributes such as the scheduled departure/arrival, the actual departure/arrival, and the departure/arrival gates. `FBPosts`[2] is a dataset of crawled Facebook posts for which we have chronological information, as well as the erroneous and the manually cleaned versions of the data (using OpenRefine [24]). The dataset contains information about a sample of posts - their title, content type, text, the week it was written, the domain and the image URL, the number of likes, and the web page it was crawled from. Missing values are the most common error type for this dataset. Both datasets have an attribute that defines the chronological order and enables splitting them into partitions. Two variants of each dataset, the one with errors occurred and the one where the errors are fixed, are provided. We utilize these variants as partitions of acceptable data quality and their corrupted counterparts for our evaluation scenario.

Datasets without ground-truth errors. `Amazon Review` [16] and the `Online Retail`[3] [9] are two retail datasets. The `Amazon Review`[4] dataset contains information about product reviews: their ID, title, category, brand, sales ranking, and related products. The `Online Retail` dataset contains historical transactional data from a UK-based retailer. It includes the invoice number, customer ID, country, quantity, description, and the unit price of a product being purchased. The third dataset contains information about `Drug Reviews`[5] [14]. It includes the name of a drug, medical conditions this drug has been designed for, ratings and reviews, the review date, and the number of users who considered this review useful. All three datasets have a mix of numeric and categorical attributes. They also contain an attribute that defines chronological order and enables partitioning, but we do not have ground-truth errors available for them.

Synthetic error types. In order to experiment with the datasets that do not provide ground truth, we inject six types of synthetic errors. We choose these types of errors because (a) they are commonly encountered in real-world use cases in industry and mentioned by many practitioners [6, 18] and (b) the majority of them is used as example error types in the research field of error detection [1, 27, 28, 34, 47]. We briefly describe these error types below.

- *Explicit missing values* - empty cells in the data as a result of wrong data collection or integration (e.g., left outer join of two tables) or, simply, an optional field in a web form that was never filled by the end-user and, thus, assigned as NULL while crawling. We remove a fraction of the values of an attribute, replacing them with NULLs;
- *Implicit missing values* - empty cells in the data that are encoded with values of an attribute's data type that semantically represent a missing value, e.g., a string 'NONE' or a numeric value out of the attribute's domain. In practice, implicit missing values are the result of missing value imputations mechanisms that are implemented in a data pipeline. We replace a fraction of the values of an attribute with 'NONE' values for textual fields or encode it as 99999 for numeric fields.
- *Numeric anomalies* - unexpected numeric values as a result of malfunctioning sensors, errors in scaling or type casting (e.g., change of measurement units from centimeters to meters, wrong parsing of *csv* files due to commas as decimal separators, etc.). For continuous numeric attributes, we corrupt a fraction of the values by replacing them with Gaussian noise that is centered at the mean value of the attribute and has a standard deviation that is scaled randomly from the interval of 2 to 5;
- *Swapped numeric fields* - misplacement of numeric values as a result of user mistake or wrong parsing, such as swapping the length and the width values of a retail product. We choose two numeric fields in the dataset and swap a fraction of the values from one attribute to another and vice versa;
- *Swapped textual fields* - analogous to swapped numeric fields on textual attributes, misplacement of textual values as a result of user mistake or wrong parsing, such as swapping the first name and the surname values of in a user registration form. We choose two textual fields in the dataset and swap a fraction of the values from one attribute to another and vice versa;
- *Typos* - unexpected spelling in textual attributes either due to user mistakes or errors in parsing (e.g., wrong encoding). We apply the "butterfinger" strategy that randomly replaces a fraction of letters in textual attributes with other letters that are neighbors on a "qwerty" keyboard layout.

Given the error types and descriptive statistics under investigation, sampling strategy does not have major effects on predictive performance of our approach in most cases. For instance, explicit

[1] http://lunadong.com/fusionDataSets.htm
[2] https://github.com/sergred/automating-data-quality-validation-data
[3] http://archive.ics.uci.edu/ml/datasets/Online+Retail/
[4] http://jmcauley.ucsd.edu/data/amazon/
[5] https://archive.ics.uci.edu/ml/datasets/Drug+Review+Dataset+%28Druglib.com%29

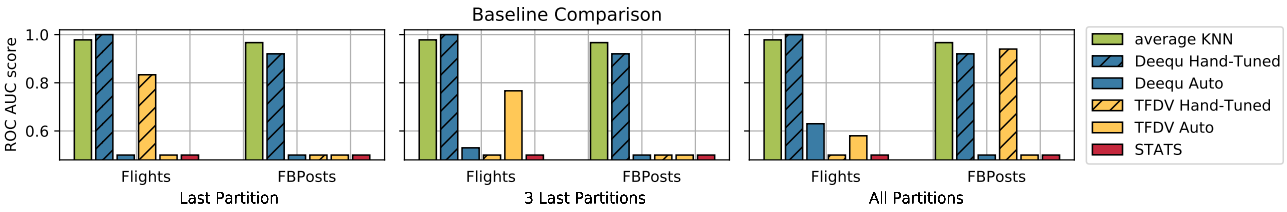


Figure 2: Comparison of the predictive performance of the proposed approach against three baseline solutions: Tensorflow Data Validation, Deequ, and statistical testing. The subplots represent three different training settings where the baselines learn from (a) only one recently observed data partition, (b) a combination of the last three data partitions, and (c) all the observed partitions. The TFDV and Deequ baselines are evaluated in their fully automated variant and a hand-tuned variant applying domain expertise. The bar chart shows that our approach outperforms the other automated baseline solutions and reaches the predictive performance of the hand-tuned baselines. The automated variants of the baselines tend to be conservative and produce false alarms in the majority of cases.

missing values would change the completeness measure, no matter wherein the data partition this error occurs. We use uniform distribution for error generation in the evaluation setup.

Hardware specification. We use an Ubuntu workstation with 8 Intel i7-8550U CPU cores (1.80GHz) and 24Gb RAM. We run all the algorithms with a single process and thread, with an exception of one baseline solution - `Deequ` library - that is built on top of Spark and runs at scale.

5.2 Comparison to Baselines

In our first experiment, we compare the predictive performance of our proposed approach ("avg. kNN" in Figure 2) to the existing baseline solutions: Tensorflow Data Validation [6], Deequ [43], and statistical testing [32, 41]. The purpose of this experiment is to evaluate whether our automated approach can reach the performance of hand-tuned state-of-the-art solutions.

Baselines. We compare the proposed approach against several existing solutions. As the first baseline, we use univariate *statistical tests* to detect shifts in data distribution between the previously observed data partitions and the current batch as an indicator of errors. We use two tests - the Kolmogorov-Smirnov test to detect shifts in continuous numeric attributes [32], and the Pearson's Chi-squared test to detect shifts in frequency distribution for categorical values [41]. For every attribute of a data partition, we run one statistical test that gives a p-value as a measure of whether the data values in the current batch come from different data distribution than the values in previously observed data partitions. We choose a test based on the attribute's data type (numerical or textual data) and compare the outcome to a common threshold of 0.05. Note that we apply Bonferroni correction to account for multiple tests.

We also use the *Tensorflow Data Validation* library [6] (TFDV) to detect data schema violations as an indicator of erroneous partitions. TFDV uses data profiling techniques to model the state of acceptable data quality by inferring their schema - attribute names, data domains, various constraints (e.g., on data distribution, uniqueness, sparsity, etc.). Then, it tests new data against inferred constraints and raises alerts upon schema violation as a signal for potential degradation of data quality. Domain experts use automated schema inference to facilitate data profiling and analysis but they have to hand-tune the schema to keep it up-to-date. In addition to the automated version of TFDV, we apply a hand-tuned version where we define its data schema based on data profiling and manual monitoring of data batches. This setting aims to compare our approach to a baseline solution that exploits domain expertise.

Lastly, we include the *Amazon Deequ* library [42] and utilize its declarative data quality constraints to validate the data. Similar to the TFDV baseline, we evaluate Deequ in both an automated variant and a hand-tuned variant. In the former, Deequ runs data profiling and constraint suggestion algorithms to generate data unit tests to validate the quality of data partitions. In the latter, we utilize a hand-tuned variant where we manually define the checks to apply based on data profiling and inspection.

Evaluation scenario. For a relational dataset d comprised of chronologically ordered partitions $d_{t_1}, \ldots, d_{t_n}$ and timestamps $t_1, \ldots, t_n$, we sequentially pick a timestamp t_k within the interval $start < k < n$, where $start$ is a predefined timestamp number to start with and n is the number of available partitions. We select $start$ as 8 in order to limit the minimum size of the training set to 8 data points. We show the partitions $d_{start}, \ldots, d_{t_{k-1}}$ as training data to each approach.

For the datasets with the ground truth, we leverage the hand-labeled "dirty" versions $\hat{d}_{t_1}, \ldots, \hat{d}_{t_n}$ of these partitions for the evaluation. We give both the clean data partition d_t and its corrupted counterpart $d_{\hat{t}}$ to each approach, and let it decide whether the data batch is of acceptable quality or contains errors. In this experiment, we use only the datasets with available ground truth to compare the predictive performance in real-world cases with unspecified error types, error magnitudes, and real-world temporal changes in data characteristics.

For each approach, we record two predictions at each timestamp t_k in the interval $start < t < n$ - one label for the partition d_t and for the erroneous counterpart $\hat{d}_t$ respectively. We compute the ROC AUC score based on the recorded prediction labels and the ground truth, where d_t has the "inlier" label, and $\hat{d}_t$ has the "outlier" label. We evaluate the automated baseline solutions in three different settings, where the automated inference is based on (a) the last, (b) three last, and (c) all previously observed partitions with no further alteration of the derived rules, constraints, or patterns, to ensure systematic comparison of our approach in a fully automated mode. With the first two settings (one and three data partitions), we evaluate whether using only the most recent data is sufficient for the automated baseline solutions to learn the state of "acceptable" data quality accurately and fast. In contrast, the third setting is applied in order to evaluate the predictive performance of baselines that take the whole training set into account and include "far-in-the-past" data partitions.

For the given experimental scenario and datasets, we spent approximately two hours per dataset for data profiling, manual inspection, and configuration of Deequ and TFDV via programming interfaces. For Deequ, we implemented declarative unit tests for data. For TFDV, we adjusted thresholds to allow for particular fractions of previously unseen data and specified data ranges. We must point out, however, that hand-tuning involved analysis of the ground-truth clean data. In this way, we simulated a "domain expert" who knows what errors are expected in the data. In real-world use cases that assume "zero domain knowledge", the analysis we conducted might be infeasible.

Results. Figure 2 depicts the comparison of the predictive performance of our approach ("Average KNN", green) against the three baseline solutions: Tensorflow Data Validation (yellow), Deequ (blue), and statistical testing (red). The bar charts report predictive performance on the `Flights` and the `FBPosts` datasets under three different training settings. The baselines learn from (a) only one recently observed data partition ("Last Partition", left), (b) a combination of the last three data partitions ("3 Last Partitions", center), and (c) all the observed partitions ("All Partitions", right). Tensorflow Data Validation and Deequ baselines are evaluated in both the fully automated mode and in their hand-tuned variant.

The results indicate that our approach outperforms other automated baseline solutions and reaches the predictive performance of hand-tuned baselines (ROC AUC score of 95%, whereas the hand-tuned Deequ solution reaches 100% and 92% on the Flights and FBPosts datasets, respectively). Other automated solutions tend to produce false alarms in the majority of cases. We attribute this to the fact that the automated baseline solutions are "conservative" and strict in terms of their chosen constraints, and thereby produce false alarms in the majority of cases.

Table 3 depicts average execution times for both our approach and the baselines. It shows that, on average, our approach is at least one order of magnitude faster than the baseline solutions. High computational efficiency is associated with the fact that both the descriptive statistics and the KNN algorithm are easy to compute and train. Since the *Deequ* library is built on top of Spark, this baseline takes more time to check data quality metrics for small datasets due to the large overhead for parallel computation. However, we assume that *Deequ* might be more efficient on large-scale data, where other baseline solutions would perform reasonably slower.

Discussion. The errors in the dataset are mostly missing values or inconsistencies due to data integration (e.g., different datetime formats for different records). To be precise, 95% of the arrival and departure time information have an inconsistent date-time format, with a large fraction of the data missing. Inconsistencies in the datetime format lead to two problems - either the year is omitted, in which case several data preprocessing techniques replace the missing value with the default year 1970, or the day and month values are swapped as the solution has no means of distinguishing these values. 63% of the arrival and departure gates information is inconsistent in the following ways: (1) presence of explicit and implicit missing values; (2) the missing value encoding differs (e.g., '-', '–', 'Not provided by airline'); or (3) the information is semantically incomplete (e.g., the 'Gate 2' value is replaced with the value 'Terminal 8, Gate 2', etc.). Since the cleaned version of the dataset was provided semi-automatically, most of the records which contained missing values were imputed where possible (e.g., by aggregation) or omitted as there were no means to guarantee the correct missing value imputation scheme. 18% of the categorical attribute 'contenttype' have implicit missing value 'nan' or syntactic mismatch in categories (e.g., a combination of German and English words for 'article'). 16% of the attribute 'text' have the wrong encoding.

Our approach performs well on the given datasets and reaches a ROC AUC score of 95%. Many of the baseline solutions, however, perform on the level of random guessing. Further analysis reveals that these baselines label the majority of the data partitions as erroneous (See Table 4). The reason why the data partitions are labeled as erroneous is due to the conservative default settings of the baseline solutions, as they are primarily designed to strictly detect data quality degradation and have false alarm rates as a secondary concern. Further analysis indicates that TFDV presumably detects errors in attributes where we know for certain

Table 3: Average execution time (in seconds) for baseline comparison. We compare our approach (Avg. KNN) against three baselines (Deequ, Tensorflow Data Validation, and statistical testing), each of them computed in three modes, where (a) one last, (b) three last, and (c) all previously observed partitions are used for training. The table shows that the average execution time of our approach is one order of magnitude faster than the baselines.

Candidate	Mode	*Flights* Data	*FBPosts* Data	*Amazon* Data
Avg. KNN	-	0.042 +- 0.001	0.006 +- 0.001	0.215 +- 0.087
Deequ	1 Last	0.322 +- 0.018	0.313 +- 0.020	0.782 +- 0.358
	3 Last	0.381 +- 0.026	0.329 +- 0.022	1.560 +- 0.800
	All	1.115 +- 0.382	0.468 +- 0.084	6.937 +- 5.427
TFDV	1 Last	0.141 +- 0.043	0.036 +- 0.008	6.679 +- 3.380
	3 Last	0.295 +- 0.060	0.058 +- 0.014	7.479 +- 3.753
	All	1.388 +- 0.702	0.126 +- 0.060	14.40 +- 9.940
STATS	1 Last	0.189 +- 0.025	0.160 +- 0.035	11.30 +- 3.575
	3 Last	0.194 +- 0.067	0.189 +- 0.061	20.20 +- 6.613
	All	0.204 +- 0.069	0.379 +- 0.439	105.6 +- 30.80

Table 4: Confusion matrices for the baseline comparison. Analogous to Table 3, we compare our approach against three baselines in three different modes. We evaluate TFDV and Deequ baselines in their fully automated variant and a hand-tuned variant applying domain expertise.

		Flights Data				*FBPosts* Data			
Candidate	Mode	TP	FP	FN	TN	TP	FP	FN	TN
Avg. KNN	-	**30**	**0**	**1**	**29**	**52**	**0**	**5**	**47**
Deequ	1 Last	**30**	0	30	0	50	2	51	1
	3 Last	**30**	0	28	2	**52**	0	52	0
	All	**30**	0	22	8	**52**	0	52	0
Deequ Hand-Tuned	1 Last	**30**	0	0	**30**	**48**	4	4	**48**
	3 Last	**30**	0	0	**30**	**48**	4	4	**48**
	All	**30**	0	0	**30**	**48**	4	4	**48**
TFDV	1 Last	0	30	0	**30**	0	52	0	**52**
	3 Last	24	6	8	22	0	52	0	**52**
	All	28	2	23	7	0	52	0	**52**
TFDV Hand-Tuned	1 Last	21	9	2	28	0	52	0	**52**
	3 Last	0	30	0	**30**	0	52	0	**52**
	All	0	30	0	**30**	**50**	2	4	**48**
STATS	1 Last	0	30	0	**30**	0	52	0	**52**
	3 Last	0	30	0	**30**	0	52	0	**52**
	All	0	30	0	**30**	0	52	0	**52**

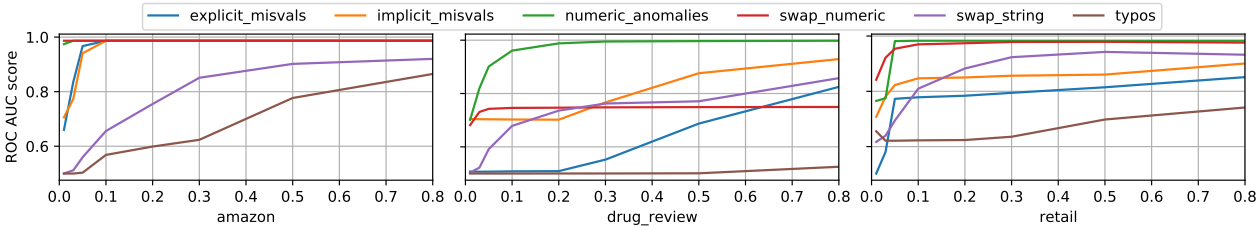


Figure 3: Overview of the predictive performance of our approach on three real-world datasets with synthetically generated errors under varying error magnitude (X-axis, 1 to 80%). We consider six error types: explicit and implicit missing values, numeric anomalies, typos in textual attributes, swapped fields for numeric and textual attributes. We observe two patterns: (a) similar predictive performance regardless of the fraction of errors (flat lines), or (b) gradual growth of the predictive performance towards bigger error magnitude, with the distinctive, more rapid growth for fractions up to 20%.

that there are no errors present. The 'Source' and the 'Flight' attributes of the *Flights* dataset do not contain errors. However, TFDV detects a violation of data schema as there are previously unseen values in the new batch, so the attribute domain has changed. A similar situation holds for the *FBPosts* dataset, with one additional type of alert - "non-boolean values" (as *FBPosts* contains one boolean attribute).

As for the hand-tuned baselines, Deequ reaches a perfect ROC AUC score on the Flights dataset and 92% on the FBPosts with hand-tuned thresholds for the completeness metric. For TFDV, the ROC AUC score ranges from 50 to 82%. The "min domain mass" parameter (i.e., a minimal fraction of data records that have to be included from the inferred data domain) was set to 0 in order to allow for any fraction of previously unseen values in the data partition. Thresholds for the completeness metric were set similarly to the Deequ baseline. This finding highlights that manual data quality monitoring and hand-tuning of existing solutions with the domain expertise is highly dataset-specific and tedious.

Note that, for Tensorflow Data Validation in several settings, the automated variants perform better than the hand-tuned variant. The reason is that the automated variants are retrained after a new data partition becomes available, whereas the hand-tuned variant is specified once on the initial training set (i.e., t_1 to t_{start}).

5.3 Sensitivity to Different Error Types and Magnitudes

In this experiment, we evaluate whether our approach detects all error types under varying error magnitudes with the similar predictive performance or whether there are error types that are harder to detect than others.

Evaluation scenario. For every dataset d with synthetically generated ground truth, we fix the error type and the error magnitude for generating corrupted data partitions $\hat{d}_t$. Other than that, the evaluation scenario is identical to the one in Section 5.2.

Results. Figure 3 shows line charts that represent predictive performance of our proposed approach per dataset and error type, where the x-axes of the plot depict the error magnitude. We are interested in the relationship between the predictive performance of our approach and the fraction of errors that are introduced in data partitions. Two distinctive patterns arise in terms of the curve shapes: (a) flat lines represent similar predictive performance regardless of the fraction of errors, whereas (b) the curves with gradual growth towards more significant error magnitudes mean that it is easier to detect degradation in data quality with greater fractions of the data partition being affected. The latter curves capture rapid increase for smaller fractions of 1 to 20%. The relative difference in predictive performance between the error types varies among the datasets and error magnitudes. Even though the `Drug Review` and the `Online Retail` datasets show resemblance in terms of the ROC AUC score, the `Amazon` dataset exhibits different patterns. For instance, the kNN novelty detection approach shows constant predictive performance rate on `Amazon`'s numeric anomalies but has a "learning curve" for `Drug Review` or `Online Retail`.

Discussion. The figure shows that, in general, the predictive performance differs from one error type to another. We attribute this behavior to two findings from the experiment's analysis. First, some types of errors are, in fact, easier to recognize than others. That statement holds for the use cases of manual data quality monitoring that are conducted by domain experts. For instance, an explicit missing value (e.g., a NULL value) is reasonably straightforward to detect even when few data records are corrupted. Other error types, such as numeric anomalies, can be detected only in cases where the ranges of acceptable values are available, or the assumption on data distribution exist [18]. Comparing ROC AUC scores between the error types, error magnitudes, and datasets indicates that predictive performance is dataset-specific and likely depends on scales and domains of every data attribute. In the majority of cases, however, missing values and numeric anomalies can be detected relatively reliably and result in high ROC AUC scores.

For every error type that we investigate, there are descriptive statistics that provide better features for classification. For instance, the completeness measure is more descriptive to detect explicit missing values. Data distribution measures(e.g., mean, standard deviation, minimum, maximum) are more descriptive to detect numeric anomalies. However, there is no single metric that is more descriptive than others for all given error types.

Note that our approach often performs reasonably well in cases of small error magnitudes (already at 10%), when introduced errors drastically affect the descriptive statistics of a data partition. Should our approach be insensitive to a specific error distribution (or particular error types), our approach can be extended by adding another descriptive statistic that is sensitive to this error distribution or error type.

Based on Figure 3, typos (brown) appear to be the hardest error type that we consider in this study. We assume that the index of peculiarity for textual attributes is a direct proxy for this error. However, predictive performance on the `Drug Review` dataset nearly reaches the level of random guessing, whereas on other datasets it exhibits a slow learning curve. Further experimental analysis reveals several differences between textual attributes on

the datasets under investigation. Our approach performs well in cases where attributes have categorical values with rather low cardinality and high repetition of values (e.g., country code). It also performs well on long texts such as reviews and descriptions with high a likelihood of word repetition within the data batch. In this case, a typo that is introduced in one word that repeats itself in the data batch yields high chances for this error to be detected by our approach, as this word becomes "peculiar" in the context of the data batch. On the other hand, typos that are introduced in almost-unique words that belong to a dictionary of a textual attribute would not be detected as this error replaces one unique word to another. For several curves that involved textual attributes, there exists a downward trend at the beginning when the training set is small. It happens due to our design decision to keep a constant contamination parameter (see Section 4, "Modeling Decisions"). In cases of small training sets, the kNN algorithm learns a broad decision boundary that leads to false positive results (i.e., where the majority of data points are considered inliers). Only with the growing training set, the decision boundary becomes smaller and yields more accurate results. One preventative measure is to ensure large initial training sets. When this is not possible, another option is to adaptively select larger contamination parameters for smaller training sets.

We obtain several findings regarding the relationship between the predictive performance of our approach and error magnitude. In general, we note two patterns in the curve. The first case is where the ROC AUC score remains approximately constant across all error magnitudes and does not depend on the fraction of corrupted records in a data partition. This happens in cases where a few erroneous records in a data partition are sufficient to affect descriptive statistics and reliably identify the data partition as erroneous. The second case is where the ROC AUC score increases gradually with the growth of the error fraction. In this case, the reason is that detecting data quality degradation becomes easier when more data in the partition are corrupted. One example is the explicit missing values error type. Note that, for this example, a clean partition d_t might allow for missing values, so that a simple rule of "100% completeness" is not applicable. Thus, the higher the difference between the fraction of missing values in between clean and erroneous data partitions, the higher is the overall ROC AUC score. Note that the shape of the curve and the rate of growth are dataset-specific.

5.4 Sensitivity to a Combination of Errors

We also extend the experiment from Section 5.3 to evaluate the sensitivity of our approach to scenarios where a combination of two different error types occurs in the same data partition.

Evaluation scenario. For every dataset d with synthetically generated ground truth, we fix the error magnitude to 50% for generating corrupted data partitions $\hat{d}_t$. We choose an attribute A_m of every data partition d_t and apply a pair of error types (if suitable for the attribute's data type). We use all pairwise combinations of error types under investigation. Other than that, the evaluation scenario is identical to the one in Section 5.2. Note that, as we sample the values-to-corrupt uniformly, there is an overlap in selected cells of a data partition d_t for the first and the second error type of the pair ($\sim$ 40%). For the overlapping values, the second error type overrides the changes made by the first type, resulting in approximate distribution of corrupted values to be 20% of the data partition and 30% respectively. In the case when the union of changes provided by each error type exceeds 50% of the data partition, we uniformly sample changes from the union to ensure total error magnitude of 50%. We compare the predictive performance of our approach to the respective performance when only one of the error types is applied.

Results. For every attribute of every dataset and every applied pair of error types under investigation, we computed three ROC AUC scores: the one where only the first error type is applied to corrupt the data, the one where only the second error type is applied, and one for a combination of applied error types. For all computed scores, we report the mean squared error of 0.028 between the ROC AUC score on a combination of error types and the maximum of ROC AUC scores where only one of the two error types is applied.

Discussion. The results indicate that the predictive performance of our approach in the case when two error types are combined is, on average, close to the performance on a single error type, the "easiest to detect" of the two, taking into account reduced error magnitudes (i.e., when errors corrupt 20-30% of the data partition separately, adding up to a total error magnitude of 50%). We generalize this observation to a combination of more than two error types that corrupt a data partition together.

5.5 Detection Quality over Time

In this experiment, we evaluate the detection quality of our approach over time. The motivation behind this experiment is twofold: (a) the size of the training set for the novelty detection algorithm grows continuously, which might gradually improve predictive performance, and (b) data characteristics are volatile and can change over time, which might lead to the occasional degradation of predictive performance.

Evaluation scenario. For every dataset d with synthetic errors, we fix the error type for generating corrupted data partitions $\hat{d}_t$. We compute two labels for every daily-ingested data partition, one for the clean variant and one for the corrupted counterpart. When we visualize ROC AUC scores over time, we aggregate these labels on a monthly basis and plot line charts with months as X-axes. Other than that, we leverage a setup that is identical to previous experiments.

Results. Figure 4 depicts the line charts that represent changes in the predictive performance of our approach over time, where the x-axis is the monthly time window (for clarity reasons, it is shown by year in the "Drug Review" graph). Two distinctive patterns arise in terms of the curve shape: (a) flat lines represent approximately constant predictive performance, whereas (b) curves with the gradual increase indicate improvements over time and, respectively, with the growing size of the training set (see `Drug Review`). The latter examples converge to a stable rate and further resemble the behavior of approximately constant predictive performance.

Discussion. The results indicate how the predictive performance of our approach changes over time, with the corresponding growth of the training set for the novelty detection algorithm to learn from. Similar to the previous section, we see two patterns. First, in most of the cases, the average prediction performance does not change significantly over time. This finding might be counter-intuitive at first, as we usually assume that an ML-based algorithm tends to perform better with more data points to train from. The reason for the approximately constant ROC AUC score is that data points that represent erroneous data partitions are likely to be far from the decision boundaries learned by the kNN

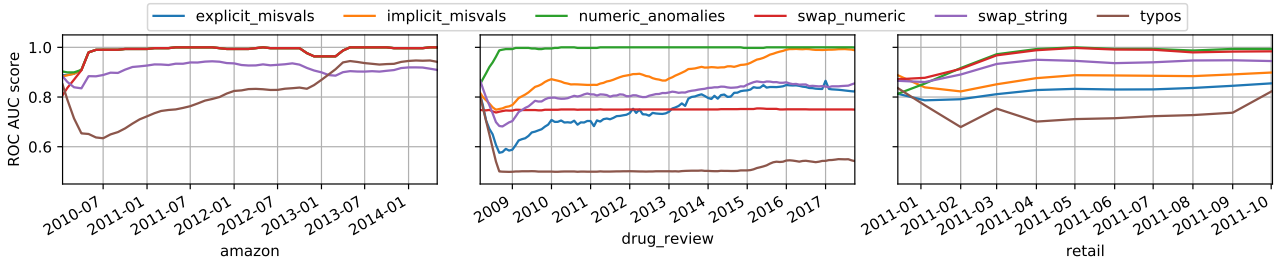


Figure 4: Predictive performance of our approach over time (X-axis). The figure depicts line charts that represent the ROC AUC score for every dataset with the ground truth, per error type over time. Various magnitudes of errors and data attributes are aggregated. The results show that, in the majority of cases, the predictive performance does not improve significantly over time with the growth of the size of the training set. Several cases (Retail, swapped fields and numerical outliers) demonstrate an initial increase of the ROC AUC score, followed by convergence to a stable rate over time. Note that, for the Amazon dataset, line charts that depict missing values, numeric anomalies, and swapped fields error types overlap and are represented by one line (red).

algorithm. These far-off data points (i.e., outliers) are likely to be detected reliably even under the limited size of the training set and, therefore, lead to the stable predictive performance of the kNN novelty detection approach. The second pattern is a gradual increase in the predictive performance in the beginning until we converge to a stable performance rate. Examples are explicit missing values and swapped textual fields for the *Drug Review* dataset. We attribute this pattern to the insufficient size of the training set to learn the decision boundaries that lead to reliable predictions. We assume that the stage of gradual increase in predictive performance corresponds to the "learning process" of the approach to derive accurate decision boundaries with clear benefits of accumulating more data points to the training set. After convergence, re-training of the approach is necessary to self-adapt to temporal changes in data.

The importance of batch frequency. Preliminary experiments show that, when choosing between daily, weekly, and monthly ingestion frequencies, daily ingestion of data led to relatively higher predictive performance. We associate this phenomenon with larger sizes of the training set.

6 RELATED WORK

We distinguish two lines of research that address related data management issues at different angles - *error detection for data cleaning* [1, 27, 36, 40, 47] and *data quality validation* [3, 20, 43].

Error detection for data cleaning. The goal of error detection mechanisms is to find the exact data records and attributes that contain errors. Abedjan et al. [1] consider four different categories of error detection solutions: (a) rule violation, (b) pattern violation, (c) outlier detection, or (d) duplicate conflict resolution based systems. Some of the algorithms require rules or patterns to be specified by the end-user. Outlier detection based methods require clean data to be present in order to "learn" what the inliers are and then decide whether particular records deviate from the expected behavior. Our approach follows similar ideas but constructs feature vectors based on the corresponding data quality metrics that are computed over the data partition instead of relying on the raw data itself. This leads to feature vectors of low dimensionality, fast model training, and guarantees numeric representation of feature vectors. The last category, duplicate conflict resolution systems, handles the specific case of duplicate entities in the data, and does not cover other types of errors. Compared to the existing error detection algorithms, our approach can detect unspecified error types and does not require domain expertise in terms of rules, patterns, or labeled examples.

Data validation. These methods aim to make a decision whether the data is valid w.r.t. particular assumptions. Tensorflow Data Validation [6] models the state of acceptable data quality with the user-defined data schema - attribute names, data domains, various constraints (e.g., on data distribution, uniqueness, sparsity, etc.). It, then, tests new data against the specified constraints and raises alerts upon schema violation. To assist the end-user, initial data schema can be inferred automatically by analyzing reference data (i.e., an "acceptable" data sample). As stated by the authors, schema refinement by domain experts is required to guarantee the performance of the library, and the schema inference functionality is provided as an aid, not a replacement of the domain expert. Data linter [20], on the contrary, validates data against data lints - deviations from accepted practices of data analysis (analogous to code lints - snippets of code that depict deviation from best practices in software engineering). The lints are predefined by the developers of the tool yet are extensible in case customized practices are in place. Another example is the Deequ library for automating the verification of data quality at scale [42], which proposes unit tests for data - a declarative specification of integrity constraints, such as completeness, consistency, syntactic and semantic accuracy, which the end-user needs to specify. Schelter et al. [42] also introduce functionality for automated constraint suggestion based on data profiles (collected descriptive statistics on data attributes). However, this method requires the presence of reference data - a sample of the data population that is considered to be of acceptable quality and is designed to generate suggestions that are validated by a domain expert. The Metanome platform [36] is a tool for data profiling that incorporates numerous algorithms for the detection of functional, order, or inclusion dependencies, as well as cardinality estimation. This method is not a data validation solution as such, but allows to automatically discover patterns from data that later could be used as rules for data quality. Metanome requires "acceptable" data samples to be present for reliable mining of the data quality patterns. As the main purpose of Metanome is data profiling and not directly data quality validation, this framework requires additional rules for detection of data quality issues and cannot be used directly as a data quality validation tool [10].

To summarize, existing approaches require domain knowledge to define rules, denial constraints, patterns, configuration of error detection solutions [1, 27], integrity constraints, data unit tests [42], error generators [39], data schema [6], or data lints [20]. Automation tools exist for data profiling, constraint suggestion, schema inference, or error detection. These solutions assume a domain expert in the loop. Our approach, in contrast, does not require any domain knowledge specified explicitly for common error types. In contrast to existing solutions, it is inspired by the work of Bleifuß et al. [4] on exploring changes in dynamic data, Ehrlinger et al. [10] on automating data quality validation, and Ioannou et al. [21] on generating benchmark data. Finally, as our experimental analysis indicates, few automated solutions for data quality validation appear to be particularly "conservative" and produce false alarms in the majority of cases.

7 CONCLUSION & FUTURE WORK

Data quality validation is crucial for large-scale production pipelines. Challenging cases are the ones where domain expertise is incomplete and data changes over time. We showed that collecting simple descriptive statistics over the data and analyzing them with novelty detection methods makes it possible to distinguish critical errors in data. In contrast to existing solutions, our approach does not require domain experts to define rules or labeled examples, and self-adapts to temporal changes in the data characteristics. We evaluated our approach against existing baselines on five real-world datasets with real and synthetically generated errors. We found that our approach detects the unspecified errors in many cases under varying error magnitudes, outperforms other automated solutions in terms of predictive performance, and reaches the ROC AUC score of baselines that are hand-tuned with domain expertise.

In future work, we plan to investigate more exotic types of errors and intend to look deeper into specific types of errors that are hard to capture by common data quality metrics, e.g., errors that are a deterministic function of the inputs (like accidentally changing the encoding). As there exist few real-world datasets that are available for evaluation purposes in data quality validation on dynamic data, we also intend to provide a set of benchmarking datasets. These datasets should contain a wide range of error types and patterns of temporal changes in data characteristics. This will enable research on controlling the false alarm rates for novelty detection algorithms in data quality validation settings.

Acknowledgements. This work was funded by the HEIBRiDS graduate school, with the support of the German Ministry for Education and Research as BIFOLD, BBDC 2 (01IS18025A), BZML (01IS18037A), the Software Campus Program (01IS17052), and Ahold Delhaize. All content represents the opinion of the authors, which is not necessarily shared or endorsed by their respective employers and/or sponsors.

REFERENCES

[1] Ziawasch Abedjan et al. 2016. Detecting Data Errors: Where Are We and What Needs to Be Done?. In *PVLDB*, Vol. 9.
[2] Fabrizio Angiulli and Clara Pizzuti. 2002. Fast outlier detection in high dimensional spaces. In *PKDD*.
[3] Dennis Baylor et al. 2017. TFX: A Tensorflow-based Production-scale Machine Learning Platform. In *KDD*.
[4] Tobias Bleifuß et al. 2018. Exploring Change - A New Dimension of Data Analytics. In *PVLDB*, Vol. 12.
[5] Michael Brackett and Production Susan Earley. 2009. The DAMA Guide to The Data Management Body of Knowledge (DAMA-DMBOK Guide). (2009).
[6] Eric Breck, Marty Zinkevich, Neoklis Polyzotis, Steven Whang, and Sudip Roy. 2019. Data Validation for Machine Learning. *SysML*.
[7] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys (CSUR)* 41, 3 (2009).
[8] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *ICALP*.
[9] Daqing Chen, Sai Laing Sain, and Kun Guo. 2012. Data mining for the online retail industry: A case study of RFM model-based customer segmentation using data mining. (2012).
[10] Lisa Ehrlinger and Wolfram Wöß. 2017. Automated data quality monitoring. In *ICIQ*.
[11] Martin J Eppler. 2006. *Managing information quality: Increasing the value of information in knowledge-intensive products and processes*. Springer Science & Business Media.
[12] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA*.
[13] Hector Garcia-Molina. 2008. *Database systems: the complete book*. Pearson Education India.
[14] Felix Gräßer, Surya Kallumadi, Hagen Malberg, and Sebastian Zaunseder. 2018. Aspect-Based Sentiment Analysis of Drug Reviews Applying Cross-Domain and Cross-Data Learning. In *DH*.
[15] Xiaoyi Gu, Leman Akoglu, and Alessandro Rinaldo. 2019. Statistical Analysis of Nearest Neighbor Methods for Anomaly Detection. In *NeurIPS*.
[16] Ruining He and Julian McAuley. 2016. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*.
[17] Alireza Heidari, Joshua McGrath, Ihab Ilyas, and Theodoros Rekatsinas. 2019. HoloDetect: Few-Shot Learning for Error Detection. In *CoRR*. arXiv:1904.02285
[18] Joseph M Hellerstein. 2008. Quantitative data cleaning for large databases. *UNECE*.
[19] Peter J Huber. 1992. Robust estimation of a location parameter. In *Breakthroughs in statistics*. Springer.
[20] Nick Hynes, D Sculley, and Michael Terry. 2017. The data linter: Lightweight, automated sanity checking for ml data sets. In *NIPS MLSys Workshop*.
[21] Ekaterini Ioannou, Nataliya Rassadko, and Yannis Velegrakis. 2013. On generating benchmark data for entity matching. *Journal on Data Semantics* (2013).
[22] Jin Huang and C. X. Ling. 2005. Using AUC and accuracy in evaluating learning algorithms. *TKDE* 17, 3 (2005).
[23] Hans-Peter Kriegel, Matthias Schubert, and Arthur Zimek. 2008. Angle-based outlier detection in high-dimensional data. In *SIGKDD*.
[24] Tien Fabrianti Kusumasari et al. 2016. Data profiling for data quality improvement with OpenRefine. In *ICITSI*.
[25] Xian Li, Xin Luna Dong, Kenneth Lyons, Weiyi Meng, and Divesh Srivastava. 2012. Truth finding on the deep web: Is the problem solved?. In *PVLDB*, Vol. 6.
[26] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *ICDM*.
[27] Mohammad Mahdavi et al. 2019. Raha: A configuration-free error detection system. In *SIGMOD*.
[28] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning. In *PVLDB*, Vol. 13.
[29] Zelda Mariet, Rachael Harding, Sam Madden, et al. 2016. Outlier detection in heterogeneous datasets using automatic tuple expansion. (2016).
[30] Markos Markou and Sameer Singh. 2003. Novelty detection: a review—part 1: statistical approaches. *Signal processing* 83, 12 (2003).
[31] Markos Markou and Sameer Singh. 2003. Novelty detection: a review—part 2: neural network based approaches. *Signal processing* 83, 12 (2003).
[32] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951).
[33] Robert Morris and Lorinda L Cherry. 1975. Computer detection of typographical errors. *IEEE Transactions on Professional Communication* 1 (1975).
[34] Felix Neutatz, Mohammad Mahdavi, and Ziawasch Abedjan. 2019. ED2: A Case for Active Learning in Error Detection. In *CIKM*.
[35] Stephen M Omohundro. 1989. *Five balltree construction algorithms*. International Computer Science Institute Berkeley.
[36] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. 2015. Data Profiling with Metanome. In *PVLDB*, Vol. 8.
[37] Neoklis Polyzotis et al. 2017. Data Management Challenges in Production Machine Learning. In *SIGMOD*.
[38] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. 2000. Efficient algorithms for mining outliers from large data sets. In *SIGMOD*.
[39] Sergey Redyuk et al. 2019. Learning to Validate the Predictions of Black Box Machine Learning Models on Unseen Data. In *HILDA*.
[40] Theodoros Rekatsinas, Xu Chu, Ihab Ilyas, and Christopher Ré. 2017. Holoclean: Holistic data repairs with probabilistic inference. In *PVLDB*, Vol. 10.
[41] Albert Satorra and Pete M Bentler. 1994. Corrections to test statistics and standard errors in covariance structure analysis. (1994).
[42] Sebastian Schelter et al. 2018. Automating Large-scale Data Quality Verification. In *PVLDB*, Vol. 11.
[43] Sebastian Schelter et al. 2019. Unit Testing Data with Deequ. In *SIGMOD*.
[44] Bernhard Schölkopf et al. 2000. Support vector method for novelty detection. In *NeurIPS*.
[45] Michael Stonebraker and Ihab Ilyas. 2018. Data Integration: The Current Status and the Way Forward. *IEEE Data Eng. Bull.* 41, 2 (2018).
[46] David Martinus Johannes Tax. 2001. *One-class classification: Concept learning in the absence of counter-examples*. Ph.D. Dissertation. TU Delft.
[47] Larysa Visengeriyeva and Ziawasch Abedjan. 2018. Metadata-Driven Error Detection. In *SSDBM*.

Provenance-Based Algorithms for Rich Queries over Graph Databases

Yann Ramusat
DI ENS, ENS, CNRS, PSL University & Inria
Paris, France
yann.ramusat@ens.fr

Silviu Maniu
Université Paris-Saclay, LRI, CNRS
Gif-sur-Yvette, France
silviu.maniu@lri.fr

Pierre Senellart
DI ENS, ENS, CNRS, PSL University & Inria & IUF
Paris, France
pierre@senellart.com

ABSTRACT

In this paper, we investigate the efficient computation of the provenance of rich queries over graph databases. We show that semiring-based provenance annotations enrich the expressiveness of routing queries over graphs. Several algorithms have previously been proposed for provenance computation over graphs, each yielding a trade-off between time complexity and generality. Here, we address the limitations of these algorithms and propose a new one, partially bridging a complexity and expressiveness gap and adding to the algorithmic toolkit for solving this problem. Importantly, we provide a comprehensive taxonomy of semirings and corresponding algorithms, establishing which practical approaches are needed in different cases. We implement and comprehensively evaluate several practical applications of the problem (e.g., shortest distances, top-k shortest distances, Boolean or integer path features), each corresponding to a specific semiring and algorithm, that depends on the properties of the semiring. On several real-world and synthetic graph datasets, we show that the algorithms we propose exhibit large practical benefits for processing rich graph queries.

1 INTRODUCTION

Graph databases [32] are part of the so-called NoSQL DBMS ecosystem, in which the information is not organized by strictly following the relational model. The structure of graph databases is well-suited to representing some types of relationships within the data, and their potential for distribution makes them appealing for applications requiring large-scale data storage and massively parallel data processing. Natural example applications of such database systems are social network analysis [13] or the storage and querying of the Semantic Web [5].

Graph databases can be queried using several general-purpose navigational query languages, an abstraction of which is *regular path queries* (RPQs) [6] (or generalizations thereof, such as C2RPQs) on paths in the graph. Recently, based on existing solutions to querying property graphs – such as Neo4j's Cypher [17] query language or Oracle's PGQL [38] – an upcoming international standard language for property graph querying, GQL [22], is being designed as a standalone language complementing SQL. GQL will notably incorporate support for RPQs.

In parallel with these recent developments, the notion of *provenance* of a query result [34], a familiar notion in relational databases, has recently been adapted to the context of graph databases [31], using the framework of provenance semirings [18]. In this framework, edges of a graph are annotated, in addition to usual properties, with elements of a semiring; when evaluating a query, traversing the paths on the graph can generate new annotations depending on the semiring operators, resulting in a semiring value associated with every query result, called the provenance of the query result. By choosing different semirings, different information on the query result can be computed. For example, when edges are annotated with elements of the *tropical* semiring (nonnegative real numbers) expressing the distance between vertices, the provenance of the query result computes the shortest distance of paths that produce this result; when edges are annotated with elements of the *counting* semiring (natural integers) interpreted as multiplicity, the provenance of the query result computes the (possibly infinite in case of cycles) number of ways each query result can be obtained. Underlying properties of the semiring directly control how the information on graph edges is encoded, and also how efficient algorithms for query processing are.

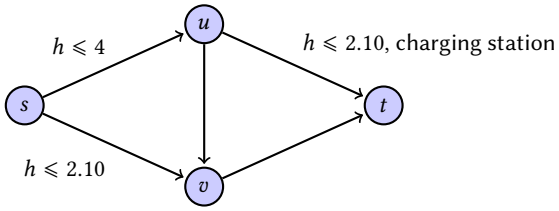


Figure 1: Example road network represented by a graph with provenance annotations along two dimensions: maximum height h (as a positive number) a vehicle must have to use the road segment, and a Boolean indicating the presence of an electrical charging station. When a dimension is not mentioned, the annotations are assumed to be, respectively, $h \leqslant \infty$ and $\neg$(charging station).

Beyond these simple examples of semirings, the framework of semiring provenance also allows modeling of intricate issues, e.g., when the problem of interest can be decomposed into several subproblems and when the resulting provenance does not necessarily correspond to a particular path in the graph.

Example 1.1. Consider the example of a road transportation network modeled as a directed graph with provenance annotations on edges. We can for example encode the presence of points of interests (such as gas stations, restaurants, or electrical charging stations) as Boolean features on edges, and road properties (e.g., maximal height or weight for a bridge or tunnel) as real-valued features.

We will show that, using semiring provenance, we can deal with graph queries that take into account multiple such features: a pair of vertices is valid for the queries if there exists at least one valid path for each restriction between the two locations. An application of this would be to ensure that different vehicle categories (say, a high-clearance truck and an electric car that requires charging on the way) can properly reach a common destination from the same origin.

Another possible semantics for semiring provenance is to check that all paths between two vertices verify (or exclude) some properties (e.g., absence of tolls, or presence of gas stations on the route) thus providing road administrators crucial information on the global state of the roads between two points.

This is illustrated in Figure 1, a road network where some road segments have restrictions on the height on vehicles; this is a first dimension of provenance. The second dimension records whether there exists an electrical charging station on the road segment – in our example, this is the case for only one edge.

In our previous preliminary research [31], we generalized three existing algorithms from a broad range of the computer science literature to compute the provenance of regular path queries over graph databases, in the framework of provenance semirings. Together, these three generalizations cover a large class of semirings used for provenance, each yielding a trade-off between time complexity and generality. We also performed experiments suggesting these approaches are complementary and practical for various kinds of provenance indications, even on a relatively large transport network.

In this paper, we extend this work by:

- Introducing a novel algorithm, MultiDijkstra, for commutative *0-closed* (or *absorptive*) semirings. This algorithm, generalizing Dijkstra's algorithm and leveraging properties of distributive lattices, partially bridges a strong computational gap between two classes of semirings left untreated in our previous research. The complexity of the queries exemplified here belongs in this gap, and strongly motivated our interest to develop the algorithms in this paper. The experiments we performed demonstrate that our new algorithm can scale up to very large networks with dozens of millions of nodes, bringing a notable improvement with respect to the state of the art of provenance computation in graph databases.
- Establishing a precise summary, in the form of a taxonomy, of the algorithms used in our context, along with their complexities and expected properties of the underlying semirings used for the provenance annotations. We also analyze similarities with classes of semirings which are used either for computing provenance of relational algebra queries [19] or of Datalog programs [11].
- Performing a comprehensive set of experiments on real-world data demonstrating the running time of provenance computation over graphs, over a wide variety of semirings and covering different use cases. We also observe that parameters depending on the topology of the graph, such as *treewidth* [27] seem to have a higher impact on the efficiency of the algorithm than distance-based parameters such as the *highway dimension* [4]. The implementation of all algorithms we use for these experiments is freely available at https://bitbucket.org/smaniu/graph-provenance/src/master/.

The paper is organized as follows. We start by introducing in Section 2 some preliminaries: graph databases enhanced by provenance annotations, a short overview of the algebraic theory of semirings, and an explanation on which semiring can be used for provenance annotations in a few selected practical applications. We revisit in Section 3 the algorithms we proposed in [31] and discuss their limitations. Section 4 is a taxonomy summarizing classes of semirings and associated algorithms for graph provenance. In Section 5, we introduce MultiDijkstra and the mathematical theory behind distributive lattices, which MultiDijkstra relies on. We present experimental results comparing all algorithms in practice in Section 6 before discussing related work in Section 7.

2 PRELIMINARIES

The framework we are considering is that of graph databases enriched with semiring-based provenance annotations. We detail here the notation and definitions we previously introduced in [31] and extend it with some additional concepts. We also introduce a large number of example semirings, to illustrate the generality of the problem considered.

2.1 Semirings

The framework for provenance in relational databases introduced by [18] uses the algebraic structure of *semirings* to encode meta-information about tuples and query results. In what follows, we present the basic notions needed for this paper; for further details about the theory and applications of semirings, see [20] and [18, 34] for their applications to provenance.

Definition 2.1 (Semiring). A *semiring* is an algebraic structure $(\mathbb{K}, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ where $\mathbb{K}$ is some set, $\oplus$ and $\otimes$ are binary operators over $\mathbb{K}$, and $\mathbb{0}$ and $\mathbb{1}$ are elements of $\mathbb{K}$, satisfying the following axioms:

- $(\mathbb{K}, \oplus, \mathbb{0})$ is a *commutative monoid*: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, $a \oplus b = b \oplus a$, $a \oplus \mathbb{0} = \mathbb{0} \oplus a = a$;
- $(\mathbb{K}, \otimes, \mathbb{1})$ is a *monoid*: $(a \otimes b) \otimes c = a \otimes (b \otimes c)$, $\mathbb{1} \otimes a = a \otimes \mathbb{1} = a$;
- $\otimes$ distributes over $\oplus$: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$;
- $\mathbb{0}$ is annihilator for $\otimes$: $\mathbb{0} \otimes a = a \otimes \mathbb{0} = \mathbb{0}$.

Example 2.2. It is easy to check that the following structures are all semirings:

Tropical semiring. $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$.

Top-k semiring. For $k \geqslant 1$ some integer,

$$((\mathbb{R}^+ \cup \{\infty\})^k, \min^k, +^k, (\infty, \ldots, \infty), (0, \infty, \ldots, \infty)),$$

where

$$\min^k((a_1, \ldots, a_k), (b_1, \ldots, b_k)) = \min^k\{a_1, \ldots, a_k, b_1, \ldots, b_k\}$$

returns the k smallest entries (with duplicates) among those in a and b, in increasing order, and

$$(a_1, \ldots, a_k) +^k (b_1, \ldots, b_k) = \min^k\{a_i + b_j \mid 1 \leqslant i, j \leqslant k\}.$$

We further impose that only tuples that are in increasing order are valid elements of the semiring. Note that the top-1 semiring is the same as the tropical semiring.

Example: For $k = 2$, $(1,2) \oplus (1,3) = \min^2\{1,1,2,3\} = (1,1)$ and $(1,2) \otimes (1,3) = \min^2\{1+1, 1+3, 2+1, 2+3\} = (2,3)$.

Counting semiring. $(\mathbb{N} \cup \{\infty\}, +, \times, 0, 1)$, where

$$\forall a \in \mathbb{N}^* \quad a + \infty = a \times \infty = \infty \times a = \infty$$

and $0 + \infty = \infty$, but $0 \times \infty = \infty \times 0 = 0$.

Boolean semiring. $(\{\bot, \top\}, \vee, \wedge, \bot, \top)$, where $\bot$ (resp, $\top$) is interpreted as the Boolean false (resp., true) value.

k-feature semiring. For $k \geqslant 1$ some integer,

$$((\mathbb{R}^+)^k, \min, \max, (\infty, \infty, \infty), (0, 0, 0))$$

where min and max are applied pointwise; it also exists in dual form, with min and max exchanged.

Integer polynomial semiring. $(\mathbb{N}[X], +, \times, 0, 1)$ where X is a finite set of variables, and $+, \times, 0, 1$ have their standard interpretations as polynomial operators and polynomial values.

Shortest-path semiring.

$$((\mathbb{R}^+ \cup \{\infty\}) \times \Sigma^*, \oplus, \otimes, (\infty, \varepsilon), (0, \varepsilon))$$

with the following operators $\oplus$ and $\otimes$:

- $(d, \pi) \oplus (d', \pi') = (\min(d, d'), \pi'')$ where π'' is π if $d < d'$, π' if $d > d'$, and $\min(\pi, \pi')$ (in lexicographic order, assuming some order on Σ) if $d = d'$;
- $(d, \pi) \otimes (d', \pi') = (d + d', \pi \cdot \pi')$ if neither d nor d' is ∞; and $(d, \pi) \otimes (d', \pi') = (\infty, \varepsilon)$ if either d or d' is ∞.

As we shall see further, these examples all yield useful applications for provenance over graphs.

We now consider properties of semirings that will be of interest to develop specific algorithms – we will illustrate these properties on the example semirings of Example 2.2. Some of the properties are summarized in Figure 2; ignore annotations for algorithms in blue for now.

A semiring is *commutative* if for all $a, b \in \mathbb{K}$, $a \otimes b = b \otimes a$. A semiring is *idempotent* if for all $a \in \mathbb{K}$, $a \oplus a = a$. In an idempotent semiring we can introduce a *natural order* defined by $a \sqsubseteq b$ iff it exists $c \in \mathbb{K}$ such that $a \oplus c = b$.[1] Note that this order is compatible with the two binary operations of the semiring: for all $a, b, c \in \mathbb{K}$, $a \sqsubseteq b$ implies $a \oplus c \sqsubseteq b \oplus c$ and $a \otimes c \sqsubseteq b \otimes c$. An important property that we wish to use in our setting is that of *k-closedness* [29], i.e., a semiring is k-closed if:

$$\forall a \in \mathbb{K}, \bigoplus_{i=0}^{k+1} a^i = \bigoplus_{i=0}^{k} a^i.$$

Here, by a^i we denote the repeated application of the $\otimes$ operation i times, i.e., $a^i = \underbrace{a \otimes a \otimes \cdots \otimes a}_{i}$. 0-closed semirings (i.e., those in which $\forall a \in \mathbb{K}, \mathbb{1} \oplus a = \mathbb{1}$) have also been called *absorptive*, *bounded*, or *simple* depending on the literature. Note that any 0-closed semiring is idempotent (indeed, $a \oplus a = a \otimes (\mathbb{1} \oplus \mathbb{1}) = a \otimes \mathbb{1} = a$) and therefore admits a natural order.

Example 2.3. All semirings in Example 2.2 are commutative except for the shortest-path semiring (indeed, concatenation is not a commutative operation).

All of them are idempotent, except for the top-k, counting, and integer polynomial semirings.

The natural order of the tropical semiring is the total order $\geqslant$ (note that this is the *reverse* of the standard order on $\mathbb{R}^+ \cup \{\infty\}$).

The tropical, Boolean, k-feature, and shortest-path semirings are 0-closed. The top-k semiring is $(k-1)$-closed. The counting and integer polynomial semirings are not k-closed for any k.

Star semirings [14], also known as *closed semirings*, extend semirings with a unary * operator, having the following property: $a^* = \mathbb{1} \oplus (a \otimes a^*) = \mathbb{1} \oplus (a^* \otimes a)$. Note that, in 0-closed semirings, we necessarily have $a^* = \mathbb{1}$. Similarly, in k-closed semirings, we can define $a^* = \bigoplus_{i=0}^{k} a^i$.

Example 2.4. As just mentioned, since the tropical, Boolean, k-feature, and shortest-path semirings are all 0-closed, we can simply define in all of them $a^* = \mathbb{1}$. Since the top-k semiring is $(k-1)$-closed, we can define a^* with the formula $a^* = \bigoplus_{i=0}^{k} a^i$.

[1] In general semirings, this defines a preorder; antisymmetry of this relation can be shown when the semiring is idempotent.

In the counting semiring we can introduce a star operator with: $0^* = 1$ and $a^* = \infty$ for $a \neq 0$.

It is not possible to simply add a star operator to the integer polynomial semiring (indeed, if the equation $x^* = 1 + (x \times x^*)$ had a solution x^* as a polynomial in x, its degree would be different on the left- and right-hand sides of the equation). However, one can define a more general semiring, that of *formal power series*, in which a star operator can be defined. See [18] for details on the semiring of formal power series, which are not important here.

We will later use the fact that a 0-closed semiring which is also *multiplicatively idempotent* (i.e., in which $a \otimes a = a$ for every a) turns out to satisfy the axioms of *bounded distributive lattices* [8, Theorem 10].

Example 2.5. The only 0-closed semirings that are multiplicatively idempotent from Example 2.2 are the Boolean and k-feature semirings.

2.2 Graph databases with provenance

We now introduce the notion of provenance in graph databases.

Definition 2.6 (Graph Database). A *graph database with provenance indication* (V, E, λ, w) over some semiring $(\mathbb{K}, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ is an edge-labeled directed graph (V, E, λ) together with a *weight function* $w : E \to \mathbb{K}$.

Given an edge $e = (u, v) \in E$, we denote $n[e] = u$ its *destination* (or next) vertex, and $p[e] = v$ its *origin* (or previous vertex). By analogy we write its weight $w[e]$ instead of $w(e)$. Given a vertex $v \in V$, we denote by $E[v]$ the set of edges having v as origin.

A path $\pi = e_1 e_2 \cdots e_k$ in G is an element of E^* with consecutive edges: $n[e_i] = p[e_{i+1}]$ for $i = 1, \ldots, k-1$. We extend n and p to paths by setting $p[\pi] := p[e_1]$, and $n[\pi] := n[e_k]$. A cycle is a path starting and ending at the same vertex: $n[c] = p[c]$. The weight function w can also be extended to paths by defining the weight of a path as the result of the $\otimes$-multiplication of the weights of its constituent edges: $w[\pi] := \bigotimes_{i=1}^{k} w[e_i]$; this can in fact be extended to any finite set of paths by $w[\{\pi_1, \ldots, \pi_n\}] := \bigoplus_{i=1}^{n} w[\pi_i]$. For any two vertices x and y of a graph G, we denote by $P_{xy}(G)$ the set of paths from x to y.

Definition 2.7 (Path Provenance). Let G be a graph database with provenance indication over some semiring $\mathbb{K}$. The *provenance between x and y*, for x and y two vertices of G is defined as the (possibly infinite) sum:

$$\text{prov}_{\mathbb{K}}(G)(x, y) := w\left[P_{xy}(G)\right] = \bigoplus_{\pi \in P_{xy}(G)} w[\pi].$$

Several problems can be defined based on this. Given two vertices s and t, the *single-pair provenance* problem computes the provenance between s and t. Given a vertex s, the *single-source provenance* problem computes the provenance between s and each vertex of the graph. Finally, the *all-pairs provenance* problem computes the provenance for all pairs of vertices.

A *regular path query* (RPQ) [6] defines a set of admissible paths from some vertex s through a regular language over edge labels. The notion of single-source provenance can be generalized to that of RPQ provenance in a straightforward manner, as we did in [31]. We also showed in [31] that computing such a provenance could be reduced in polynomial time to the single-source provenance problem; this works by constructing a product of the graph with

the automaton describing the language of the query. Note that this construction can be done on-the-fly (avoiding generation of inaccessible vertices) and that the size of the automaton is usually quite small; thus, the overhead is usually affordable oven for large graphs as showed experimentally in [31]. We will implicitly use this reduction throughout the paper, meaning that we only need to consider the single-source provenance problem in the rest of the paper. Consequently, we will also ignore edge labels and see a graph database as defined by its vertices, edges, and semiring weights.

2.3 Semantics of path provenance

As defined, the provenance between two vertices in a graph database is in fact a (possibly infinite) sum over the provenances of all paths from the source vertex leading to the target vertex. As we observed in [30], the only possible source of non-finiteness in the sum is due to cycles in the graph, so that we only need to be able to sum all the powers of a given semiring value. For this to be semantically meaningful we need the semiring to be a star semiring, and we additionally need the star operator to verify for all semiring element a: $a^* = \bigoplus_{n=0}^{\infty} a^n$ for some well-behaved infinitary sum operation $\bigoplus$ (namely, associativity, and distributivity of $\otimes$ over this infinitary sum operator). This class of semirings is commonly known as *countably complete star semirings*, *c-complete star semirings* [24], or *ω-complete star semirings*.

Example 2.8. All star semirings identified in Example 2.4 are, indeed, c-complete star semirings. Note that, for k-closed semirings, the infinitary sum $\bigoplus_{n=0}^{\infty} a^n$ is simply $\bigoplus_{n=0}^{k} a^n$, and the condition of being a c-complete star semiring is trivially satisfied by our choice of star operator. In the remaining cases (counting semiring, formal power series, formal language semiring) one can verify that a well-behaved infinitary sum operation can be introduced, and that it verifies $a^* = \bigoplus_{n=0}^{\infty} a^n$.

We also pointed out in [30] that all-pairs graph provenance is equivalent to the computation of the *asteration* of the matrix corresponding to the graph representation with provenance tags as cell-values. With all these definitions in place, we observe that the semantics of provenance over specific semirings actually corresponds to a various number of problems of interest. Remember that using the construction of [31] we can extend this to the provenance of arbitrary RPQs.

Example 2.9. Let G be a graph database over some c-complete star semiring $\mathbb{K}$, and s and t fixed source and target vertices in G. The provenance between s and t corresponds to the following notions, depending on the semiring $\mathbb{K}$:

- **Tropical semiring:** length of shortest path between s and t.
- **Top-k semiring:** lengths of k shortest paths between s and t.
- **Counting semiring:** total number of paths between s and t, edge weights being interpreted as number of edges between two vertices.
- **Boolean semiring:** existence of a path between s and t, depending on the existence of edges denoted by their Boolean weights.
- **k-feature semiring:** minimum feature value along each dimension of all paths between s and t; if min and max are exchanged, maximum feature value along some path from s to t.
- **Formal power series:** how-provenance, see [18].
- **Shortest-path semiring:** pair formed of a length l and path label π such that π is the shortest path from s to t, of length l (if there are multiple shortest paths, π is the first in lexicographic order).

Example 2.10. Let us return to the example in Figure 1. We model the charging station Boolean feature as an integer feature by simply setting $\top = 1$ and $\bot = 0$. We take the (max, min) definition of the k-feature semiring where we compute the maximum value of each feature among some path from origin to destination, and we order heights in decreasing order (e.g., by taking their inverse) so that a higher feature value means a (more restrictive) lower height.

Consider two types of vehicles of interest that want to reach the vertex t from the vertex s: one has height between 3 and 4 meters, the second is a small ($h \leqslant 1.5$) electric car that needs at least one charging station on the road to t. In the presence of the edge from u to v, both of them can reach t from s; without that edge, only the electric car is able to. This is reflected in the provenance: $\mathsf{prov}(G)(s,t) = (4, \text{charging station})$ while $\mathsf{prov}(G \backslash \{(u,v)\})(s,t) = (2.10, \text{charging station})$.

3 EXISTING ALGORITHMS

We now provide a review of three algorithms to solve the single-source provenance problem, also previously described in [31]. Each of these algorithms yields a different trade-off between time complexity and applicability to various types of semirings, as summarized in Table 1.

Algorithm 1 DIJKSTRA – single-source

Input: $(G = (V, E, w), s)$ a graph database with provenance indication over $\mathbb{K}$ and the source s.
Output: Array $\mathbf{w}$ representing the single-source provenance from s of the reachability query.

```
1:  S ← ∅
2:  w[a] ← 𝕆, ∀a ∈ V
3:  w[s] ← 𝟙
4:  while S ≠ V do
5:      Select a ∉ S with minimal w[a]
6:      S ← S ∪ {a}
7:      for each neighbor b of a not in S do
8:          w[b] = w[b] ⊕ (w[a] ⊗ w[ab])
9:      end for
10: end while
11: return w
```

DIJKSTRA. Dijskstra's algorithm is generally used to solve shortest-distance problems in directed graphs. However, as shown also in [31], the algorithm readily generalizes to our semiring context, by placing some restrictions on the semirings used. For instance, the tropical semiring is exactly the semiring that allows to compute the shortest distance, as in the original algorithm. The general flow of the algorithm – using general semiring operations – is outlined in Algorithm 1, and Table 1 indicates its running time (in terms of the graph size and the costs of the semiring operations $\oplus$ and $\otimes$). Dijkstra's algorithm is known to be a very efficient algorithm. However, this efficiency comes from the fact that it uses a priority queue: once a value is extracted from it, we know that it is the correct one – this allows us to only visit each vertex in the graph once. This only works if we apply DIJKSTRA to semirings which are 0-*closed* (or absorptive) and in which an additional condition is satisfied: the natural order is a *total order* [31].

As we shall discuss later, there is a large complexity gap between DIJKSTRA on the one hand and the other two algorithms

Table 1: Required semiring properties and asymptotic complexity for each studied algorithm, where $\mathrm{T}_\bullet$ is the complexity of the elementary semiring operation $\bullet$. The last column assumes constant cost for all semiring operations.

Name	Semiring property	Time complexity (with semiring op.)	Time complexity
MatrixAsteration	star	$O(\lvert V\rvert \mathrm{T}_* + \lvert V\rvert^3(\mathrm{T}_\oplus + \mathrm{T}_\otimes))$	$O(\lvert V\rvert^3)$
NodeElimination	c-complete star	$O(\lvert V\rvert \mathrm{T}_* + \lvert V\rvert^3(\mathrm{T}_\oplus + \mathrm{T}_\otimes))$	$O(\lvert V\rvert^3)$
Mohri	k-closed	Exponential	Exponential
MultiDijkstra	0-closed $\otimes$-idempotent	$O\left(\ell \times (\mathrm{T}_\oplus \lvert V\rvert \log \lvert V\rvert + \lvert E\rvert\,(\mathrm{T}_\oplus + \mathrm{T}_\otimes))\right)$	$O(\ell \times (\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert))$
Dijkstra	0-closed total ordered	$O(\mathrm{T}_\oplus \lvert V\rvert \log \lvert V\rvert + \lvert E\rvert(\mathrm{T}_\oplus + \mathrm{T}_\otimes))$	$O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$

we discuss in this section – NodeElimination and Mohri – on the other. This is the main motivation to introduce the new algorithm we present in Section 5.

Algorithm 2 Mohri – single-source [29]

Input: $(G = (V, E, w), s)$ a graph database with provenance indication over $\mathbb{K}$ and the source s.
Output: Array $\mathbf{w}$ representing the single-source provenance from s of the reachability query.

```
1:  for i ∈ {1, ..., |Q|} do
2:      w[i] ← r[i] ← 𝕆
3:  end for
4:  w[s] ← r[s] ← 𝟙
5:  S ← {s}
6:  while S ≠ ∅ do
7:      q ← head(S)
8:      dequeue(S)
9:      r' ← r[q]
10:     r[q] ← 𝕆
11:     for each e ∈ E[q] do
12:         if w[n[e]] ≠ w[n[e]] ⊕ (r' ⊗ w[e]) then
13:             w[n[e]] ← w[n[e]] ⊕ (r' ⊗ w[e])
14:             r[n[e]] ← r[n[e]] ⊕ (r' ⊗ w[e])
15:             if n[e] ∉ S then
16:                 enqueue(S, n[e])
17:             end if
18:         end if
19:     end for
20: end while
21: w[s] ← 𝟙
22: return w
```

Mohri. Mohri [29] introduced an algorithm for computing single-source provenance for reachability queries over k-closed semirings. Outlined in Algorithm 2, it performs, in a manner similar to the Bellman–Ford algorithm, step-by-step relaxations over the edges of the graph (lines 13–14), maintaining a queue to decide in which order the elements are inspected. The queue can be chosen in different ways: based on the topology of the graph, e.g., if the graph is acyclic; or a queue prioritized by weight when, e.g., one wishes to compute top-k shortest paths using the top-k semiring.

In the worst case, the theoretical complexity of this approach is exponential in the size of the graph [29], mainly due to the fact that the algorithm may have to visit the same cycle in the graph multiple times. However, the complexity heavily depends on the implementation of the queue. For instance, for top-k shortest paths, implementing a priority queue allows for an efficient algorithm, having polynomial complexity. Indeed, as we shall detail later, for road transportation networks and top-k shortest paths, experiments show an almost linear-time behavior in k and the size of the graph.

In contrast, the algorithm may be much more inefficient in practice for other types of networks (such as social networks). As we conjecture in Section 6, this may be due to the fact that transport networks have relatively low *treewidth* [27]. The treewidth is a parameter measuring how much a graph (or more generally any relational instance) resembles a tree. Many intractable problems over graphs have tractable solutions on instances of fixed treewidth. We confirm in Section 6 that many of the algorithms for provenance computation strongly benefit – in terms of running time – from low treewidth.

Another important graph parameter – stemming from the active research community around computing routing for, e.g, driving directions – the *highway dimension* [4] has been introduced to provide a theoretical basis for the efficiency observed in practice in state-of-the-art heuristics for computing optimal transport paths. This parameter relies heavily on weights on the edges of the graphs and the distribution of shortest distances in the graph. In our experiments in Section 6, we evaluate whether this parameter also explains the practical efficiency of our algorithms for computing the provenance of routing queries.

Algorithm 3 NodeElimination – single-pair

Input: $(G = (V, E, w), s, t)$ a graph database with provenance indication over $\mathbb{K}$, the source s, and the target t.
Output: Single value $\mathbf{w}_{s't'}$ representing the single-pair provenance between s and t of the reachability query.

```
1:  V' ← V ∪ {s', t'}
2:  E' ← E ∪ {(s', s), (t, t')}
3:  for i ∈ V' do
4:      for j ∈ V' do
5:          w^(0)_ij ← { w[ij]          if i ≠ j,
                       { 𝟙 ⊕ w[ij]     if i = j
6:      end for
7:  end for
8:  for k in V do
9:      for each (p, q) s.t. (p, k), (k, q) ∈ E' do
10:         w_pq ← w_pq ⊕ (w_pk ⊗ w*_kk ⊗ w_kq)
11:     end for
12: end for
13: return w_s't'
```

NodeElimination. The most general algorithm available is based on the idea of Brzozowski and McCluskey for obtaining a formal language expression (i.e., a regular expression) equivalent to the language of an automaton [9]. The algorithm is outlined in Algorithm 3. The algorithm works by eliminating vertices one by one and computing the "shortcut" values for each vertex pair, until only the source and target vertices remain. This algorithm works for any *c-complete semiring over which a star operation is defined* – this is necessary for the shortcuts computed in the algorithm to be correct.

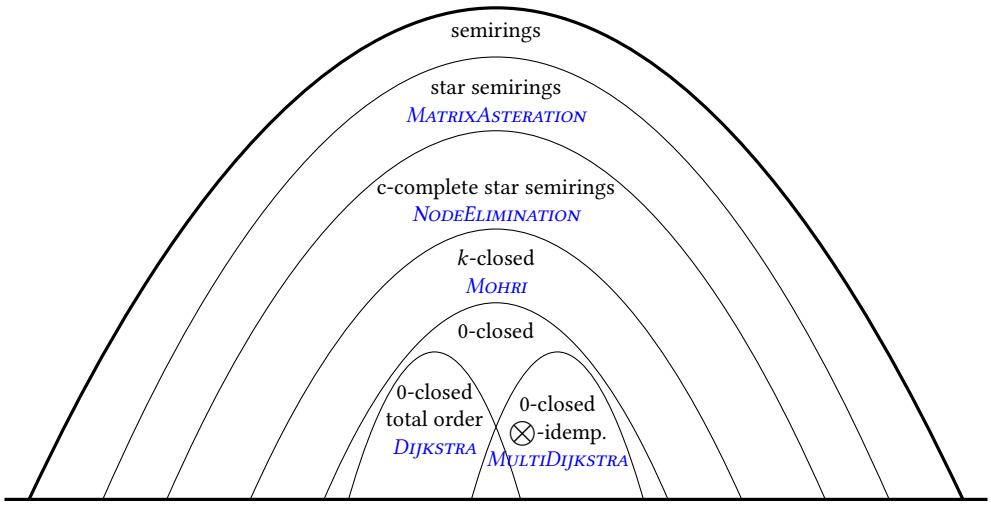


Figure 2: Taxonomy of the semirings used for graph provenance along with algorithms that work on them

In general, the complexity of the algorithm is at least cubic in the number of vertices in the graph, which makes it practically unusable on large graphs. Importantly, however, it also can be shown that its complexity is closely related to the treewidth parameter of the graph. Following a simplicial elimination order (unfortunately not tractable to compute) one can rephrase the complexity shown in Table 1 in terms of the treewidth parameter w by $O(|V|\mathrm{T}_* + |V|w^2(\mathrm{T}_\oplus + \mathrm{T}_\otimes))$. Thus, if the treewidth is small over, e.g., transportation networks, one can benefit from heuristics for finding a suitable elimination order to optimize this algorithm. We dedicate a part of our experiments demonstrating the impact of some heuristics (for instance, focusing on vertices of higher degrees) on the running time of this algorithm.

Related algorithms. Star semirings are also known as closed semirings [2] and the star operation is known as the closure operation. In this sense, all-pair computations correspond to *matrix asteration*. For instance, the NodeElimination algorithm can be used to compute the asteration [2] of a matrix – but, if the semiring is not c-complete, there is no guarantee of a semantics compatible with the intuitive semantics of provenance over graph databases. Matrix asteration allows for a high degree of parallel computation [1].

4 TAXONOMY

We present in Figure 2 a high-level view linking the properties and classes of semiring we presented in Section 2 and their associated algorithms, presented in Sections 3 and 5. The figure shows a clear hierarchy of classes of semirings, both in terms of the complexity of the algorithm and the expressive power of the semirings.

An important practical application that is similar to our setting is the provenance for Datalog queries introduced in [18] and further optimized using circuits [11]. Datalog [3] is a language derived from Prolog, useful for infering new knowledge given existing facts and a set of inference rules. In the papers above, the semiring classes for which optimization of queries is possible are strikingly similar: PosBool(X) and Sorp(X) discussed in [11, 18] correspond respectively to the positive fragment of the Boolean function semiring, and to the free (i.e., most general) 0-closed semiring. In that sense, algorithm optimizations discussed here apply directly to applications such as Datalog query optimization.

5 ALGORITHM FOR 0-CLOSED SEMIRINGS

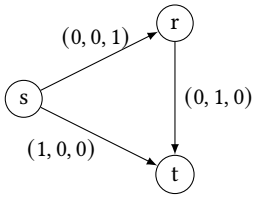
As explained in Section 3, Dijkstra requires a total natural order on the elements of a 0-closed semiring. This is quite a restrictive setting (among the examples from Example 2.2, only the *tropical semiring* fits), while using a more generally available algorithm such as Mohri can lead to practical inefficiency. The question we are addressing in this section is whether we can bridge this complexity gap and still obtain practical algorithms for 0-closed semiring without total orders.

First, we present an example semiring setting, with non-total natural order, where Dijkstra cannot be readily applied.

Example 5.1. Let us consider the 3-feature semiring

$$(\{0,1\}^3, \min, \max, (1,1,1), (0,0,0)).$$

In the example graph below, the provenance between s and t is: $\min(\max((0,0,1),(0,1,0)),(1,0,0)) = (0,0,0)$ and that between s and r is: $\min(\max((1,0,0),(0,1,0)),(0,0,1)) = (0,0,0)$



Assume there would be an order for which Dijkstra computes this provenance. Then, starting from s, Dijkstra would select either r and assign it provenance $(0,0,1)$, which is wrong, or t and assign it provenance $(1,0,0)$, which is also wrong.

In the following, we address this problem and design a new algorithm, MultiDijkstra (for *Multidimensional Dijkstra*) that applies to the more general case of 0-closed semirings for which multiplication is idempotent (such as the k-feature semiring, but also the Boolean function semiring used in probabilistic databases, see [34]). As it turns out, such semirings satisfy the axioms of *bounded distributive lattices* [8, Theorem 10]; this allows us to design an efficient algorithm for answering queries using these types of semirings.

5.1 Mathematical Background

In the following we introduce basic notions about finite distributive lattices. We assume the lattices we use are finite because

we are only ever using the subsemiring generated by edge annotations. As we shall see, this subsemiring is finite when both operations of the semiring are idempotent.

We refer the reader to [36] for more details regarding the theory behind distributive lattices.

5.1.1 Definitions and Notation. A *lattice* $(L, <)$ is a partially ordered set (*poset*) where every two elements have a unique infimum (their *meet*, $\wedge$) and supremum (their *join*, $\vee$). A *lattice embedding* of a lattice L into a lattice K is a one-to-one join and meet homomorphism from L to K. In a poset, an element y covers x (denoted $x \lessdot y$) if $x < y$ and there are no such z such that $x < z < y$. A lattice embedding ℓ is *tight* if $x \lessdot y$ implies $\ell(x) \lessdot \ell(y)$.[2]

An element x of a lattice L is *join-irreducible* if $x = a \vee b$ implies that $x = a$ or $x = b$. The set of non-zero join-irreducible elements of L is denoted $J(L)$. It induces a subposet of L which is also denoted by $J(L)$.

For a subset S of a lattice L, we let $\bigvee S = \bigvee_{x \in S} x$ be the join of the elements of S. We often write $\bigvee_L S$ to specify that the join takes place in L. A subset S of a poset is a *downset* or *ideal* if $x \in S$ and $y \leqslant x$ implies $y \in S$. The minimum downset containing an element x is denoted $\mathtt{id}\, x$. We note $\mathcal{D}(P)$, for a poset P, the family of downsets of P ordered by inclusion.

A *chain* C of length n in a poset P is a subposet isomorphic to the linear order $\mathbb{Z}_n$ on the n elements $\{0, 1, \ldots, n-1\}$. A *chain decomposition* of a poset P is a partition of its elements into a family $\mathcal{C}$ of chains $C_1, \ldots, C_d$. For a family $\mathcal{C} = \{C_1, \ldots, C_d\}$ of disjoint chains, the product $\prod \mathcal{C} := \prod_{i=1}^{d} C_i$ consists of all d-tuples $x = (x_1, \ldots, x_d)$ where $x_i \in C_i$ for each $i \in \{1, \ldots, d\}$. It is ordered by $x \leqslant y$ if $x_i \leqslant y_i$ for each i.

5.1.2 Results. A classical result from Birkhoff [7] establishes an isomorphism between L and $\mathcal{D}(J(L))$:

Theorem 5.2 ([7]). *The map* $\mathbf{S} : x \mapsto \mathtt{id}\, x \cap J(L)$ *is an isomorphism of* L *to* $\mathcal{D}(J(L))$. *Its inverse is* $S \mapsto \bigvee_L S$.

For a chain decomposition $\mathcal{C}$ of a poset, let $\mathcal{C}_0$ be the family of chains we get from the chains in $\mathcal{C}$ by adding a new minimum element to each. In [12], Dilworth proved the following embedding theorem:

Theorem 5.3 ([12]). *For any chain decomposition* $\mathcal{C}$ *of a poset* P *the map* $S \mapsto \bigvee_P S$ *is an embedding of* $\mathcal{D}(P)$ *into* $P = \prod \mathcal{C}_0$.

Then, we obtain the following corollary we will use later:

Corollary 5.4. *Given a chain decomposition* $\mathcal{C}$ *of a distributive lattice* L, *there is a tight embedding of* L *into* $\prod \mathcal{C}_0$.

5.2 Application to Provenance Computation

Corollary 5.4 provides us with a way to compute provenance over distributive lattices using a multidimensional version of Dijkstra's algorithm. Because an embedding is a homomorphism, we can compute each component of $\prod \mathcal{C}_0$ independently. And because the homomorphism is one-to-one, we can easily recover the provenance at the end of the computation.

Example 5.5. If we take a look at distributive lattice of the divisors of 60 with greatest common divisor (gcd) and least common multiple (lcm) as join and meet operators, we notice that the divisors of 60 are either powers of 2, 3, 5 or an lcm of these integers. Thus, they can be represented using three dimensions representing the factorization of 60 along these prime numbers: $\mathit{decompose}(4) = (2, 0, 0)$, $\mathit{recompose}(0, 1, 0) = 3$, and $\mathit{recompose}(2, 1, 0) = 12$. We can then compute *independently* each dimension of the result using Dijkstra's algorithm since each component is totally ordered; then, partial results are combined.

In other words, we can run separately, ℓ times, Dijkstra's algorithm for each dimension of this product, where ℓ is the number of chains in the chain decomposition. This gives us a parameterized algorithm, where ℓ depends on the semiring. For example, for the semiring used in Example 5.1, $\ell = 3$. We outline the algorithm in pseudo-code in Algorithm 4. We need the following routines that are highly specific to the semiring: $\mathit{decompose}(e)$ takes as parameter an element e of L and returns its image $v(e) \in \mathcal{P}$. For the opposite direction $\mathit{recompose}(d_1, \ldots, d_n) = \bigvee_{0 \leqslant i \leqslant n} d_i$ returns as expected an element of L.

We use as a subroutine a slightly modified version of Dijkstra, parameterized by the semiring dimension and working with semirings having elements in vector form, corresponding to the decomposition. *Dijkstra(s,t,i)* $\in J(L)$ computes the provenance between s and t corresponding to the i^{th} dimension of the decomposition.

Example 5.6. We describe the working of Algorithm 4 in the example presented in Example 5.1: first, each edge value is decomposed; this step is easy to follow as the 3-feature values are already presented in decomposed form. A second step consists in calculating values along each dimensions. Algorithm 1 is launched a first time over the graph with edge values corresponding to the first dimension: 0 for (s, r) and (r, t), 1 for (s, t). The result is 0. Algorithm 1 is launched a second time over the graph with edge values corresponding to the second dimension: 0 for (s, r) and (s, t), 1 for (r, t). The result is, again, 0. Finally, Algorithm 1 is launched a third time over the graph with edge values corresponding to the third dimension: 0 for (s, t), 1 for (s, r) and (r, t). The result is 0. This ends the second step. The third step consists in recomposing partial values obtained by successive applications of Dijkstra's algorithm. This ends up to the final provenance value of $(0, 0, 0)$.

Algorithm 4 MultiDijkstra – single-pair

Input: $(G = (V, E, w), s, t)$ a graph database with provenance indication over $\mathbb{K}$, the source s, and the target t.
Output: Single-pair provenance of the reachability query from s to t.

```
1: for each edge e ∈ E do
2:     decompose(w(e))
3: end for
4: for each dimension i do
5:     d_i ← Dijkstra(s, t, i)
6: end for
7: return recompose(d_1, ..., d_n)
```

For the sake of simplicity, we presented the single-pair version of our algorithm. To extend it to the single-source version one only needs to perform the *recompose* subroutine for each vertex in the graph.

To minimize accesses to the *decompose* subroutine – which can be very costly – we optimize MultiDijkstra by adopting a lazy approach, where the *Dijkstra* subroutine calls *decompose* only when needed, storing the decomposition across calls. This avoids scanning the whole graph when s and t are close.

[2] Implicitly from lattice notation to poset notation: $x \vee y = y$ means $x \leqslant y$.

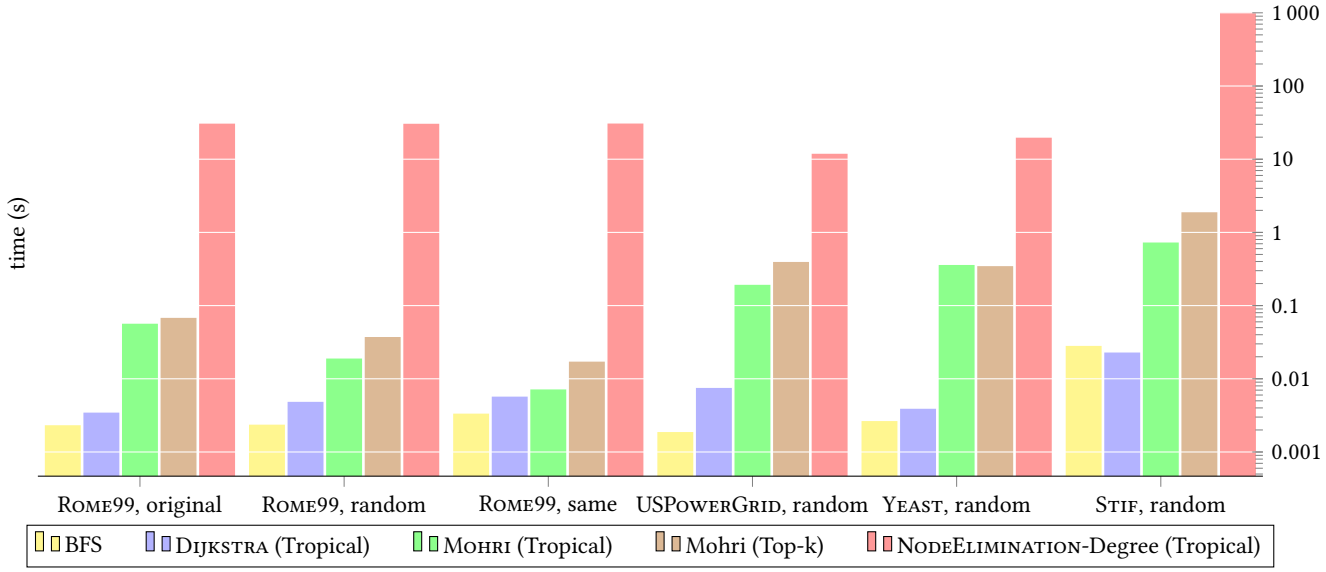


Figure 3: Comparison between algorithms for shortest distances

Table 2: Graph datasets: size and treewidth lower and upper estimates from [27]

type	name	# of vertices	# of edges	tw
infrastructure	Paris	4 325 486	5 395 531	55–521
	Stif	17 720	31 799	28–86
	USPowerGrid	4 941	6 594	10–18
	Rome99	3 353	4 831	5–50
social	Facebook	4 039	88 234	142–237
biology	Yeast	2 284	6 646	54–255

Two other optimizations implemented are a stopping condition that ends the *Dijkstra* subroutine when a visited vertex has value $\mathbb{0}$, and lazy initialization of the priority queue. These two optimizations led to vastly improved computation times over the naive implementation.

5.3 Practical Use Case

As exemplified in the Introduction, k-feature semirings can be used to ensure that all paths from s to t verify a combination of features (they all go through a specific set of points of interests, or verify some road properties) or either ensure the existence of valid paths up to some collection of restrictions. We show in the experimental section that this is tractable for practical use cases (continental-sized areas, around 10^7 vertices). To the best of our knowledge, no solution for this that scales even to graphs of thousands of vertices has been previously proposed.

6 EXPERIMENTS

We performed experiments on real-world graph data, using an Inria computing cluster running the OAR task manager. The individual vertices of the cluster have a minimum of 48 GB of RAM, and run Intel Xeon X5650 or E5-26xx CPUs.

We used datasets[3] from a variety of domains, mostly representing infrastructure networks: the OpenStreetMaps network of Paris (Paris), the Paris public transport network (Stif), and the power grid of the continental US (USPowerGrid). For comparison, we have also evaluated on other types of datasets: a small subset of the Facebook social network (Facebook) and the yeast protein-to-protein interaction network (Yeast). All these datasets come without provenance annotations, that we add in different ways depending on experiments. We also used a real weighted road transportation network dataset Rome99, with tropical semiring annotations, from the 9th DIMACS Implementation Challenge[4]. This dataset consists of a large portion of the directed road network of the city of Rome, Italy, from 1999. Basic information about the resulting graphs are summarized in Table 2.

For datasets without provenance annotations, unless specified differently, we randomly generate weights in the tropical semiring for benchmarks, uniformly between 1 and 3 000. To be able to compare the impact of the weights on the performance of the algorithms, we also use a constant-weight setting, where all weights equal to 1. Each experiment generally represents the average over 10 runs (random choices of origin and destination vertices).

Our experimental study is focused on comparing the four algorithms presented in this paper, over several semirings. We provide a comparison of all of our algorithms for the computation over the tropical semiring (shortest distance), since all algorithms can be used in this setting. We investigate the running time and the number of relaxation steps performed by Mohri and MultiDijkstra algorithm, using initial weights provided by the dataset Rome99, as well as custom weights (all identical and all random); we then study over all datasets the impact of the elimination order heuristic on the overall performance for NodeElimination. We then finish with the comparison between our new algorithm and previous solutions to demonstrate its efficiency.

Evaluating shortest distances. We start by evaluating how the algorithms deal with the shortest distance semiring, i.e., the tropical and top-k semiring (by setting $k = 1$). The properties of this semiring allow their implementation for the first three algorithms:

[3]These datasets were used in [27] for treewidth computation experiments, and are downloadable from https://github.com/smaniu/treewidth/; some of them originate from http://snap.stanford.edu/data/index.html.

[4]http://users.diag.uniroma1.it/challenge9/download.shtml

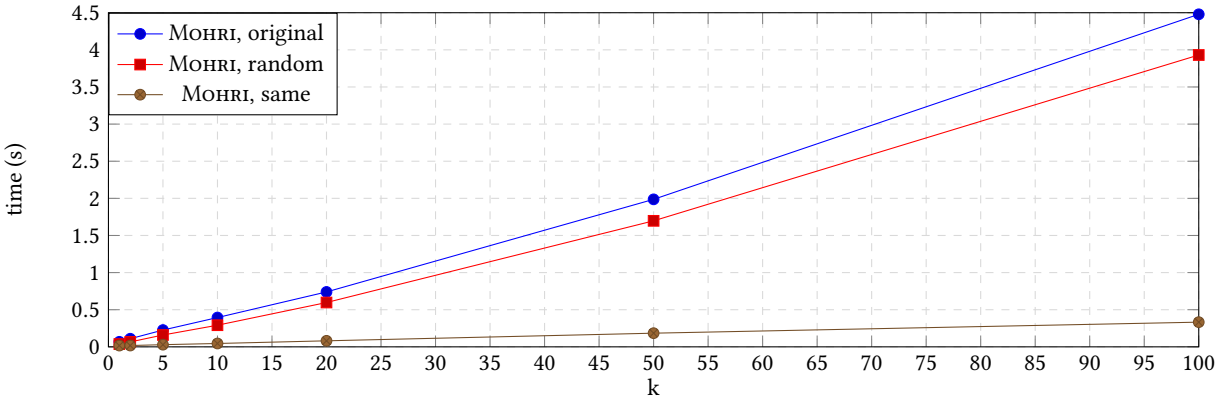


Figure 4: Computation time for Mohri over the top-k distances semiring, for varying values of k and varying weight assignments (Rome99)

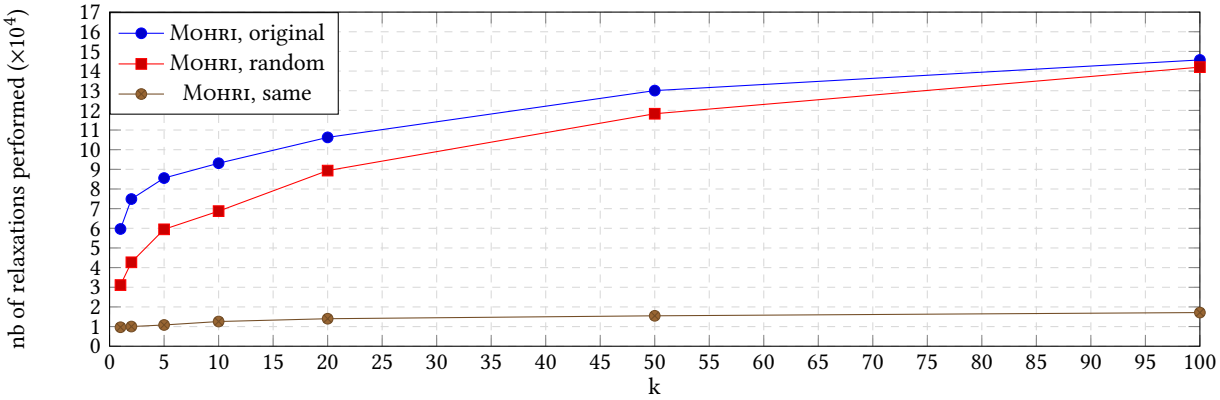


Figure 5: Number of relaxations performed by Mohri over the top-k distances semiring, for varying values of k and varying weight assignments (Rome99)

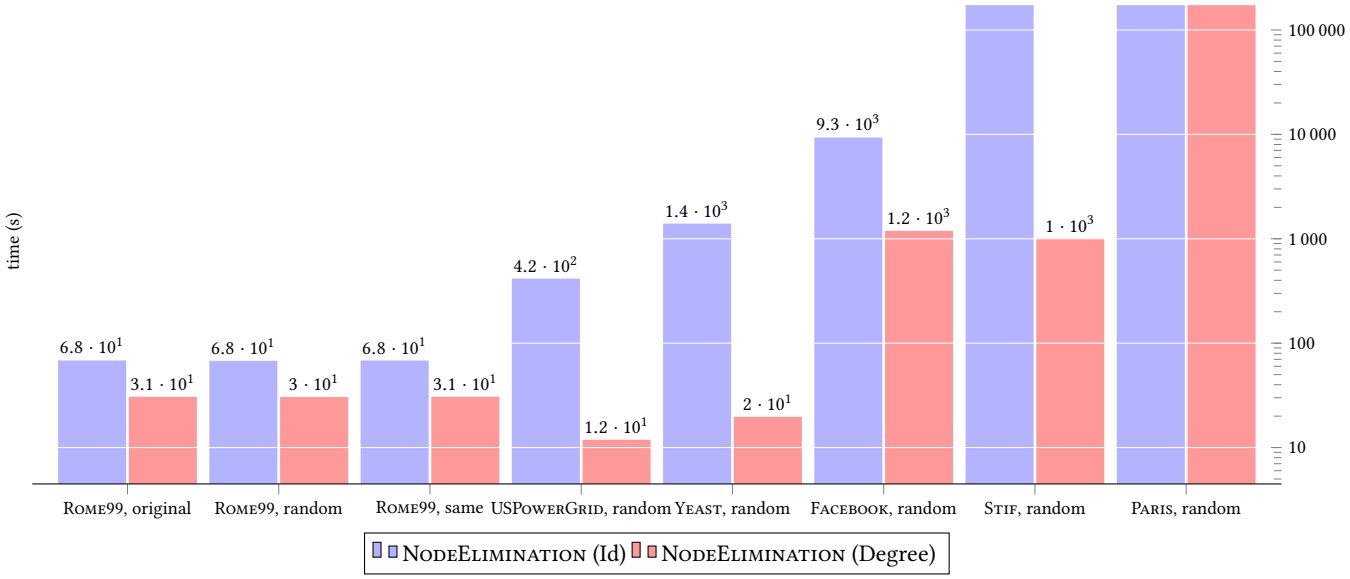


Figure 6: Comparison between elimination orders for NodeElimination algorithm (tropical semiring). Values greater than 100 000 s are timeouts.

Dijkstra, Mohri, and NodeElimination, whereas MultiDijkstra reduces to Dijkstra in that case. We also implemented a breadth-first-search traversal for computing accessibility with no provenance information (BFS). This also allows us to compare the performance of algorithms against non-annotated graph databases.

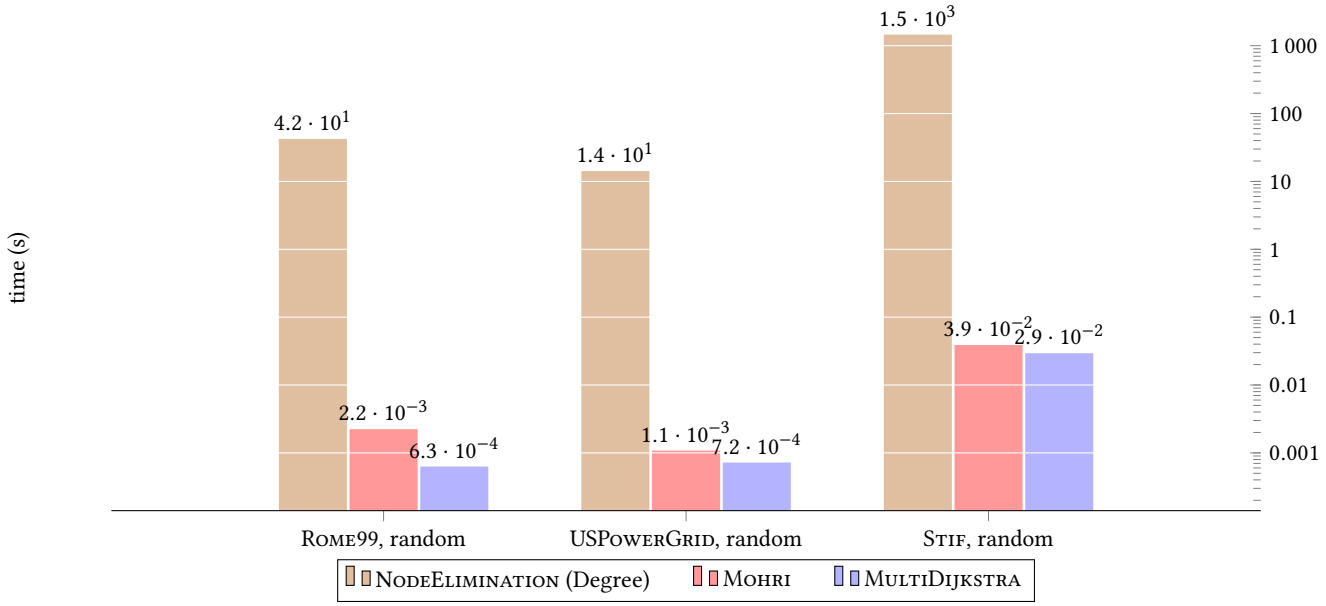


Figure 7: Comparison between NodeElimination, Mohri, and MultiDijkstra (3-feature semiring)

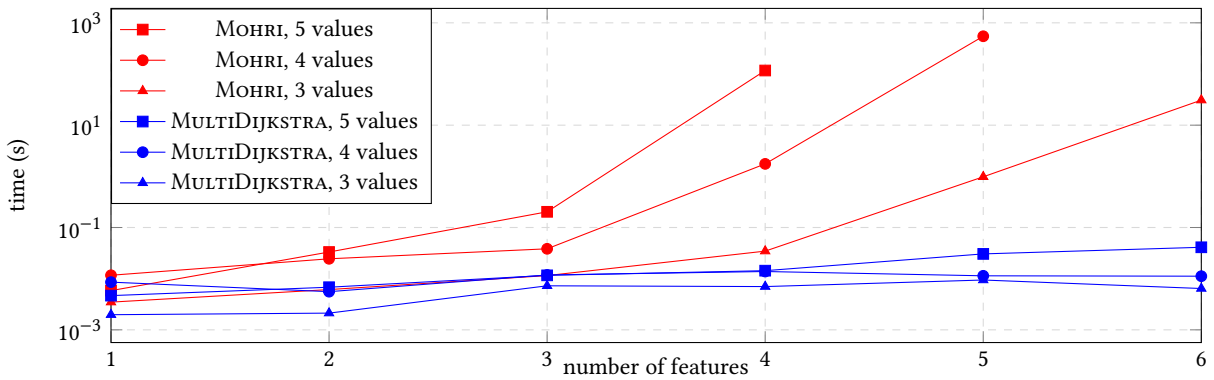


Figure 8: Computation time for Mohri and MultiDijkstra depending on the number of dimensions (Rome99)

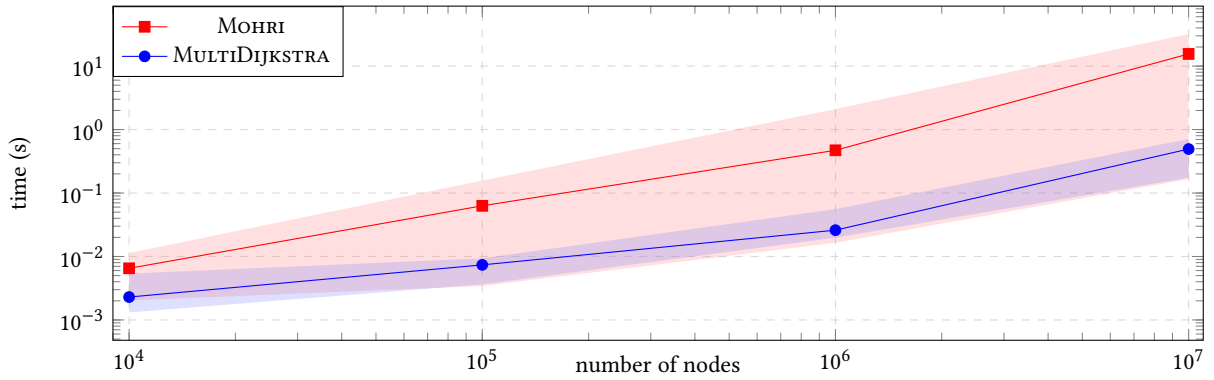


Figure 9: Average computation time for Mohri and MultiDijkstra over random graphs depending on the number of nodes; shaded areas indicate minimum and maximum computation times observed (3-feature semiring)

Figure 3 shows, on a logarithmic scale, the result for our graphs, and for some settings of weights (original, random, or same weights). It is immediately clear from the figure that the choice of algorithm is crucial: we need the most specialized algorithm for the semiring we use: Dijkstra is more efficient than Mohri which is more efficient than NodeElimination. Even for Mohri, we notice that using it configured for the top-k semiring with $k = 1$ does introduce an overhead in execution; when using the tropical semiring directly the overhead is smaller. We also show the overhead introduced when using provenance annotations is quite limited, as the difference between Dijkstra and BFS is less than an order of magnitude for each dataset, and Dijkstra

sometimes even outperforms BFS. Finally, NodeElimination is always several orders of magnitude slower than Dijkstra. Another encouraging result is that Mohri – which allows more classes of semirings than Dijkstra – has a reasonable running time in practice, despite the stated exponential complexity bound in the original paper. We turn to evaluating its performance next.

Mohri in practice. In Figure 4 and in Figure 5 we respectively study the impact of the factor k on the running time and on the number of computations performed by the algorithm. Our results show that the computational time is linear in k, though this is not the case for the number of relaxations, which increases sublinearly in k. This means that for large values of k the algorithm spends most of its time maintaining the queue.

We also compare the performance of the algorithm depending on weight assignment (original, random, same). It seems that considering random values instead of "real" values has almost no significant impact over the efficiency of the algorithm. This is a somewhat disappointing result because it rules out the possibility to parametrize the complexity of the algorithm through network parameters, for instance, in terms of the *highway dimension* [4] – a graph parameter that has been successfully applied for understanding the efficiency of state-of-the-art shortest-distance algorithms in road networks. However, the performance increases significantly when all weights are uniform, which may be expected since computation of shortest distances become far simpler, and far more paths have equal distance.

As pointed out in Section 3 this algorithm performs extremely well over transportation networks. We wanted to provide a comparison of its working time for different kinds of graphs (especially graphs whose treewidth is large relative to their size). For this purpose we used a social network dataset: who-trusts-whom network of people who trade using Bitcoin on a platform called Bitcoin Alpha [25, 26] (3 783 vertices and 24 186 edges). The algorithm times out after 48 hours.

What we can learn from this is that the key property making Mohri so efficient over transportation networks is not due to distance properties (e.g., highway dimension) – impacted by the weights of the connections – but rather by topological properties of the underlying graph (e.g., treewidth).

Ordering for NodeElimination. NodeElimination's perfomance, due to its main loop of creating "shortcuts" in the graph, is heavily dependent on the order in which the vertices are eliminated. This elimination ordering is strongly linked to the *treewidth* parameter of the graph. For instance, following a degree based elimination order gives an upper bound on this parameter.

Hence, we have compared different elimination orders for NodeElimination and found out that the minimum degree based elimination order (*Degree*) greatly improves the efficiency of this algorithm compared to having no such heuristic (*Id*). This improvement can be dramatic, as for the Yeast dataset where the algorithm is two orders of magnitude faster. As expected, weights over the edges doesn't impact the running time, as shown in Figure 6.

This is important in practice: running NodeElimination on low-treewidth graphs (e.g., infrastructure and transport networks) can be the difference between the algorithm being unusable and allowing reasonable running times. Taking into account that NodeElimination allows for a large class of semirings, this can have a significant real-world application impact.

MultiDijkstra. We now evaluate MultiDijkstra, our contribution to bridging the gap between absorptive semirings and more general ones. We compare it to Mohri and NodeElimination in the case of the k-feature semiring, which is kind of the canonical semiring that is 0-closed and multiplicatively idempotent. Figure 7 showcases this on 3 datasets. In all cases, our new algorithm is between 3 and 4 orders of magnitude faster than NodeElimination, depending on the network we use, and significantly faster than Mohri.

We then performed an additional experiment (Figure 8), examining the impact of the number of features and values actually used in each feature on the running time of both algorithms. We found out that when either one of the two criteria reaches 4, Mohri times out while MultiDijkstra keeps scaling.

Finally, Figure 9 presents a comparison between Mohri and MultiDijkstra on large Erdős–Rényi random generated graphs (generated using Python networkx's *fast_gnp_generation* method, using an average of 1.7 edges per vertex) show that our new algorithm is still tractable for continental-sized graphs of millions of vertices. Interestingly, MultiDijkstra also exhibits a much smaller variance than that of Mohri, whose performance varies by more than one order of magnitude between runs.

7 RELATED WORK

The idea of encapsulating operations carried along by graph algorithms in terms of semirings has been really common for decades. In [10, Chapter 25] the authors presented two of the classical graph algorithms, Floyd–Warshall and transitive closure algorithm in terms of *closed semirings*. The APSP (*All-Pairs Shortest-Path* problem) is elegantly expressible using star semirings; hence, research focused on the links to linear algebra through matrix computations [1], allowing to speed up the response time using parallel computations. Recent work on semiring-based graph processing has provided to the community some tools such as GraphBLAS [23], a library of kernel functions dedicated to optimize linear algebra computations over sparse matrices. Unfortunately, this tool focuses essentially on matrix and vector products and is not amenable to express priority queue management such as those needed for Mohri, Dijkstra, MultiDijkstra. Only NodeElimination and the matrix asteration algorithms could benefit of a GraphBLAS implementation: this might increase their performance, even when retaining their higher asymptotic complexity with respect to other algorithms.

Amongst many other fields, semirings have been successfully applied in constraint-solving programming [8], linguistic structure prediction [37] and formal language theory [33]. This algebraic structure is also perfectly suited to the modeling of dynamic programming [21].

The notion of provenance has also been initially developed using semirings [18], either for relational databases and Datalog programs, leading to practical systems such as [35], an extension to PostgreSQL adding the support for provenance. Many representation frameworks have been successfully applied to speed up the computation of the provenance for Datalog programs, most notably a circuit-based provenance approaches [11] and the solving of fixed-point equations using derivation tree analysis [15]. The latter approach led to a proof-of-concept implementation [16] of the resolution of fixed-point equations over *c-continuous semirings* using the Newton method.

Compared to our work, relational databases lack the effective support for navigational queries (recursion is an issue) and Datalog programs are much more expressive than graphs (they are closely related to hypergraphs), so we suspect query answering in Datalog would be highly inefficient for the continental-sized road-network datasets we target, though we leave this investigation for future work.

Numerous notions of provenance co-exist in the literature and each target different usages. The notion we use in this paper considers the provenance to be computational rather than just informational: we can apply operations over our provenance values with different semantics depending on the underlying semiring. Some practical systems, such as [28] rely on property graphs to represent provenance annotations, that are of an informational rather than computational nature. Those systems focus on the further querying of obtained provenance to derive additional information about the process.

8 CONCLUSIONS

We presented in this paper a study on evaluating the provenance of rich graph queries using the semiring provenance framework. We established a taxonomy of semiring classes, based on their properties. This in turn allows us to find, for a set of important semiring classes, the most appropriate algorithm, enabling real-world applicability. We introduce a new algorithm, MultiDijkstra, which bridges the gap between algorithms for absorptive semirings and ones for more general classes.

Experimentally, on graph datasets from various domains, we showed that making sure that the appropriate algorithm is chosen for the semiring specialization is crucial; gains of several orders of magnitude are observed between algorithms on the same graph datasets. Moreover, we notice that algorithms for which their theoretical complexity is high perform well in practice, especially on graphs having relatively low treewidth.

We believe the link with classes of semiring for which an optimization for the computation of the provenance for Datalog queries exists is a key observation for optimizing computations in our framework. Investigating this further will allows us to benefit from the rich literature around Datalog provenance (in particular, [11]) and to compare to our solutions.

ACKNOWLEDGMENTS

This work has been funded by the French government under management of Agence Nationale de la Recherche as part of the "Investissements d'avenir" program, reference ANR-19-P3IA-0001 (PRAIRIE 3IA Institute).

REFERENCES

[1] S. Kamal Abdali. 1994. Parallel Computations in *-Semirings. In *Computational Algebra*, Klaus G. Fischer, Philippe Loustaunau, Jay Shapiro, Edward L. Green, and Daniel Farkas (Eds.). Taylor & Francis, Chapter 1, 1–16.

[2] S. Kamal Abdali and David Saunders. 1985. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science* 40 (1985), 257–274. https://doi.org/10.1016/0304-3975(85)90170-7

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison Wesley.

[4] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato Fonseca F. Werneck. 2010. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *SODA*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 782–793. http://dl.acm.org/citation.cfm?id=1873601.1873665

[5] Marcelo Arenas and Jorge Pérez. 2011. Querying semantic web data with SPARQL. In *PODS*. New York, 305–316.

[6] Pablo Barceló. 2013. Querying Graph Databases. In *PODS*. ACM, New York, 175–188.

[7] Garrett Birkhoff. 1937. Rings of sets. *Duke Math. J.* 3, 3 (1937), 443–454. https://doi.org/10.1215/S0012-7094-37-00334-X

[8] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. 1997. Semiring-based constraint satisfaction and optimization. *J. ACM* 44, 2 (1997), 201–236. https://doi.org/10.1145/256303.256306

[9] Janusz A. Brzozowski and Edward J. McCluskey. 1963. Signal Flow Graph Techniques for Sequential Circuit State Diagrams. *IEEE Trans. Electr. Comp.* EC-12, 2 (1963), 67–76.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms* (2nd ed.). The MIT Press.

[11] Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. 2014. Circuits for Datalog Provenance. In *ICDT*. 201–212.

[12] Robert P. Dilworth. 1950. A Decomposition Theorem for Partially Ordered Sets. *Annals of Mathematics* 51, 1 (1950), 161–166. http://www.jstor.org/stable/1969503

[13] Pedro Domingos and Matthew Richardson. 2001. Mining the network value of customers. In *KDD*. ACM, New York, 57–66.

[14] Manfred Droste, Werner Kuich, and Heiko Vogler. 2009. *Handbook of Weighted Automata*. Springer, Berlin.

[15] Javier Esparza and Michael Luttenberger. 2011. Solving fixed-point equations by derivation tree analysis. In *International Conference on Algebra and Coalgebra in Computer Science*. Springer, 19–35.

[16] Javier Esparza, Michael Luttenberger, and Maximilian Schlund. 2014. FPsoLvE: A Generic Solver for Fixpoint Equations Over Semirings. *International Journal of Foundations of Computer Science* 26. https://doi.org/10.1007/978-3-319-08846-4_1

[17] Nadime Francis, Andrés Taylor, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, and Petra Selmer. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD*. 1433–1445. https://doi.org/10.1145/3183713.3190657

[18] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *PODS*. ACM, New York, 31–40.

[19] Todd J. Green and Val Tannen. 2017. The Semiring Framework for Database Provenance. In *PODS*. Association for Computing Machinery, New York, NY, USA, 93–99. https://doi.org/10.1145/3034786.3056125

[20] Udo Hebisch and Hanns J. Weinert. 1998. *Semirings: Algebraic Theory and Applications in Computer Science*. World Scientific, Singapore.

[21] Liang Huang. 2008. Advanced Dynamic Programming in Semiring and Hypergraph Frameworks. (2008), 18.

[22] ISO SC32 / WG3. [n.d.]. Graph Query Language GQL. https://www.gqlstandards.org/.

[23] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. https://doi.org/10.1109/HPEC.2016.7761646

[24] Daniel Krob. 1987. Monoides et semi-anneaux complets. *Semigroup Forum* 36 (1987), 323–339.

[25] Srijan Kumar, Bryan Hooi, Disha Makhija, Mohit Kumar, Christos Faloutsos, and V. S. Subrahmanian. 2018. Rev2: Fraudulent user prediction in rating platforms. In *WSDM*. 333–341.

[26] Srijan Kumar, Francesca Spezzano, V. S. Subrahmanian, and Christos Faloutsos. 2016. Edge weight prediction in weighted signed networks. In *ICDM*. 221–230.

[27] Silviu Maniu, Pierre Senellart, and Suraj Jog. 2019. An Experimental Study of the Treewidth of Real-World Graph Data. In *ICDT*. Lisbon, Portugal, 18. https://doi.org/10.4230/LIPIcs.ICDT.2019.12

[28] Hui Miao, Amit Chavan, and Amol Deshpande. 2016. ProvDB: A System for Lifecycle Management of Collaborative Analysis Workflows. *CoRR* abs/1610.04963 (2016). arXiv:1610.04963 http://arxiv.org/abs/1610.04963

[29] Mehryar Mohri. 2002. Semiring Frameworks and Algorithms for Shortest-distance Problems. *J. Autom. Lang. Comb.* 7, 3 (2002), 321–350.

[30] Yann Ramusat. 2019. Provenance-Based Routing in Probabilistic Graph Databases. In *VLDB 2019 PhD Workshop*. http://ceur-ws.org/Vol-2399/paper08.pdf

[31] Yann Ramusat, Silviu Maniu, and Pierre Senellart. 2018. Semiring Provenance over Graph Databases. In *TaPP*. https://www.usenix.org/conference/tapp2018/presentation/ramusat

[32] Ian Robinson, Jim Webber, and Emil Eifrem. 2013. *Graph Databases*. O'Reilly Media.

[33] Arto Rozenberg, Grzegorz; Salomaa. 1997. Handbook of Formal Languages || Semirings and Formal Power Series: Their Relevance to Formal Languages and Automata. Vol. 10.1007/978-3-642-59136-5. https://doi.org/10.1007/978-3-642-59136-5_9

[34] Pierre Senellart. 2017. Provenance and Probabilities in Relational Databases: From Theory to Practice. *SIGMOD Record* 46, 4 (2017).

[35] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. 2018. ProvSQL: Provenance and Probability Management in PostgreSQL. *Proceedings of the VLDB Endowment (PVLDB)* 11, 12 (Aug. 2018), 2034–2037. https://doi.org/10.14778/3229863.3236253

[36] Mark Siggers. 2014. On the representation of finite distributive lattices. *arXiv* 1412.0011 [math] (2014), 16. http://arxiv.org/abs/1412.0011

[37] Noah A. Smith. 2011. Linguistic Structure Prediction. *Synthesis Lectures on Human Language Technologies* 4, 2 (May 2011), 1–274. https://doi.org/10.2200/S00361ED1V01Y201105HLT013

[38] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *GRADES*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2960414.2960421

Sequence detection in event log files

Ioannis Mavroudopoulos
Aristotle University of Thessaloniki, Greece
mavroudo@csd.auth.gr

Theodoros Toliopoulos
Aristotle University of Thessaloniki, Greece
tatoliop@csd.auth.gr

Christos Bellas
Aristotle University of Thessaloniki, Greece
chribell@csd.auth.gr

Andreas Kosmatopoulos
Aristotle University of Thessaloniki, Greece
akosmato@csd.auth.gr

Anastasios Gounaris
Aristotle University of Thessaloniki, Greece
gounaria@csd.auth.gr

ABSTRACT

Sequential pattern analysis has become a mature topic, with a lot of techniques for a variety of sequential pattern mining-related problems. Moreover, tailored solutions for specific domains, such as business process mining, have been developed. However, there is a gap in the literature for advanced techniques for efficient detection of arbitrary sequences in large collections of activity logs. In this work, we make a threefold contribution: (i) we propose a system architecture for incrementally maintaining appropriate indices that enable fast sequence detection; (ii) we investigate several alternatives for index building; and (iii) we compare our solution against existing state-of-the-art proposals and we highlight the benefits of our proposal.

1 INTRODUCTION

Event log entries refer to timestamped event metadata and can grow very large; e.g., even a decade ago, the amount of log entries of a single day was at the order of terabytes for certain organizations, as evidenced in [3]. Due to their timestamp, the log entries can be regarded as event sequences that follow either a total or a partial ordering. The vast amount of modern data analytics research on such sequences is divided into two broad categories.

The first category comprises sequential pattern mining [11], where a large set of sequences is mined to extract subsequences that meet a variety of criteria. Such criteria range from frequent occurrence, e.g., [23, 33] to importance and high-utility [12]. In addition, there are proposals that examine the same problem of finding interesting subsequences in a huge single sequence, e.g., [25]. However, these techniques fail to detect arbitrary patterns, regardless of whether they are frequent or interesting; e.g., they are tailored to a setting where a support threshold is provided and only subsequences meeting this threshold are returned, whereas we target a different problem, that is to return all subsequence occurrences given a pattern.

The second category of existing techniques deals with detecting event sequences on the fly and comprises complex event processing (CEP). CEP is a mature field [14, 34] and supports several flavors of runtime pattern detection. We aim to solve a similar problem to CEP but tailored to a non-streaming case, where pattern queries are submitted over potentially very large log databases. Since logs continuously arrive, we account for periodic index building and we support pattern matching where the elements in the pattern are not strictly in a sequence in the logs, e.g., in the log sequence `ABBACC`, we are interested in detecting the occurrence of the pattern `ABC` despite the fact that in the original sequence other elements appear in between the elements in the searched pattern. Given that we relax the constraint of strict contiguity, techniques based on suffix trees and arrays are not applicable. Contrary to CEP, we aim to detect all pattern occurrences efficiently and not only those happening now.

In summary, our contribution is threefold: (i) we propose a system architecture for incrementally maintaining appropriate indices that enable fast sequence detection and exploration of pattern continuation choices; (ii) we investigate several alternatives for index building; and (iii) we compare our solution against existing suffix array-based proposals, focusing on logs from business processes, showing that not only we can achieve high performance during indexing but we also support a broader range of queries. Compared to other state-of-the-art solutions, like Elasticsearch, we perform more efficient preprocessing, while we provide faster query responses to small queries remaining competitive in large queries in the datasets examined; additionally, we build on top of more scalable technologies, such as Spark and Cassandra, and we inherently support pattern continuation more efficiently. Finally, we provide the source code of our implementation.

The structure of the remainder of this paper is as follows. We present the notation and some background next. In Section 3, we introduce the architecture along with details regarding preprocessing and the queries we support. We discuss the index building alternatives in Section 4. The experimental evaluation is in Section 5. In the last sections, we discuss the related work, open issues and present the conclusions.

2 PRELIMINARIES

In this section, we first present the main notation and then we briefly provide some additional background with regards to the techniques against which we compare our solution.

2.1 Definitions and notation

We aim to detect sequential patterns of user activity in a log, where a log contains timestamped events. The events are of a specific type; for instance, in a log recording the user activity on the web, an event type may correspond to a specific type of click and in business processes, an event corresponds to the execution of a specific task. The events are logically grouped in sets, termed as cases or sessions or traces[1], which may correspond to a specific session or the same process instance or, in the generic case, grouped by other user-defined criteria. More formally:

[1] In this work, we use the terms trace, case and session interchangeably.

	Our Method	Exact rooted subtree matching
Supported policy	SC, STNM	SC, Tree Matching
Database usage	Yes	No
Preprocess rationale	Indexing of all possible pairs	Indexing of all the subtrees
Query processing rationale	Combination/merging of results of pairs in the query sequence	Binary search in the subtrees space

Table 1: Differences between the technique in [19] and our method

Symbol	Short description
L	the log containing events
A	the set of activities (or tasks), i.e., the event types
E	the set of all events
C	the set of cases, where each case corresponds to a single logical unit if execution, i.e., a session, a trace of a specific business process instance execution, and so on
ev	an event ($ev \in E$), which is an instance of an event type
ts	the timestamp of an event (also denoted as $ev.ts$)
l	the size of A, $\|A\|$
n	the maximum size of a case
m	the size of C, $\|C\|$

Table 2: Frequently used symbols and interpretation

Definition 2.1. (Event Log) Let A be a finite set of activities (tasks). A log L is defined as $L = (E, C, \gamma, \delta, ts, \preceq)$ where E is the finite set of events, C is the finite set of Cases, $\gamma : E \rightarrow C$ is a surjective function assigning events to Cases, $\delta : E \rightarrow A$ is a surjective function assigning events to activities, ts records the timestamp denoting the recording of task execution and $\preceq$ is a strict total ordering over events belonging to a specific case, normally based on execution timestamps.

The notion of timestamp requires some further explanation. In sessions like web user activity and similar ones, events are usually instantaneous. However, this is not the case in task executions in business processes. In the latter case, the timestamp refers to either the beginning of the activity or its completion, but in any case, logging needs to be consistent. The duration of activities can only be estimated implicitly and not accurately from the difference between the timestamps of an event and its successor, because there may be delays between the completion of an activity and the beginning of the execution of the next activity downstream. However, systematic analysis involving task duration can be conducted only if the exact task duration is captured, which requires extensions to the definition above. Such extensions is out of the scope of this work and are orthogonal to our contributions.

Table 2 summarizes the main notation; $|A|$ is denoted as l, the maximum size of a case is denoted as n, and the size of the set of cases $|C|$ is denoted as m.

Next, to provide the context of the queries we aim to support, we define the two main types of event sequence detection that we employ in this work:

Strict contiguity (SC) , where all matching events must appear strictly in a sequence one after the other without any other non-matching events in-between. This definition is widely employed in both exact subsequence matching and CEP systems and stream processing engines, such as Flink [6].[2] For example, SC applies when we aim to detect pattern occurrences, where a search for a product on an e-shop website is immediately followed by adding this product to the cart without any other action in between.

Skip-till-next-match (STNM) , where strict contiguity is relaxed so that irrelevant events are skipped until we detect the next matching event of the sequential pattern [34]. STNM is required when, for example, we aim to detect pattern occurrences where after three searches for specific products there is no any purchase eventually in the same session.

Example: let us assume that we look for a pattern `AAB`, where `A,B` are activities. Let us also assume that a log contains the following sequence of events `<AAABAACB>`, where the timestamps are implicit by the order of each event. SC detects a pattern occurrence starting at the 2nd position, whereas STNM detects two occurrences; the first one contains the events at the 1st, 2nd and 4th position, while the second one contains the events at the 5th, 6th and 8th position. Note that other types of event sequence detection allow for additional and overlapping results, e.g., to detect a pattern in the 1st, 3rd and 8th position, as discussed at the end of this work [34].

2.2 Exact rooted subtree matching in sublinear time

Strict contiguity (SC) is directly relevant to subsequence matching and tree-based techniques have been used for a long time for finding sub sequences in large datasets. Suffix trees and suffix arrays are commonly used to this end. The method presented in [19] can find subtrees in sublinear time and it has been used to detect possible continuations of a given event sequence in business processes in [27].

In a nutshell, the technique in [19] solves the problem of finding the occurrences of a subtree with m modes in a tree T with n nodes in $O(m + logn)$, after pre-processing the tree. First, the string W corresponding to T is created; this is achieved through traversing the tree in a preorder manner and adding a 0 every time we recur to a previous level. This yields a W of length equal to $2n$. W is then used to create a suffix array A, in which the starting positions of the $2n$ suffices are specified. After discarding those starting with 0, we end up with n suffices. The main property of A is that suffices are sorted by the nodes' order. The subtree to be searched in T is first mapped to a preorder search string, and then a binary search in A is performed.

In Table 1 we present the high-level differences between this method and our proposal. We rely on simple indexing employing a database backend, while, during query processing, the main

[2] https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html

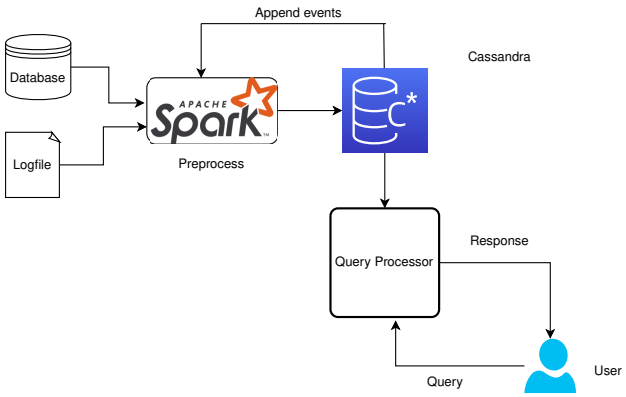


Figure 1: Architecture overview

operation is merging and post-processing of sorted lists, as explained in the next sections. More importantly, we support both STNM and SC pattern types.

3 SYSTEM ARCHITECTURE

There exist several CEP proposals along with fully-fledged prototypes and complete systems that allow users to query for Strict contiguity (SC) or Skip-till-next-match (STNM) patterns, but these are operating in a dynamic environment over a data stream. Therefore, we need to develop a system that can receive adhoc pattern queries over a large collection of logs and process them in an efficient manner. These queries will be defined later in this section and are broadly divided into three main categories (statistics, pattern detection and pattern expansion). We focus on offline pattern detection, but we account for the fact that the logs are constantly growing bigger and bigger. This entails that any practical approach needs to be incremental, i.e., to support the consideration of new logs periodically.

The overview of our proposed architecture is shown in Figure 1. There exists a database infrastructure containing old logs and, periodically, new logs are appended. There are two main components in the architecture. The pre-processing component constructs and/or updates an inverted index that is leveraged during query processing. This index is stored in a key-value database to attain scalability. In our implementation, we have chosen Cassandra[3], because of its proven capability to deal with big data and offer scalability and availability without compromising performance. However, any key-value store can be used in replacement.

The second component is the query processor, which is responsible for receiving user queries, retrieving the relevant index entries and constructing the response.

These two components are described in more detail in the remainder of this section, while indexing is discussed in the next section.

3.1 The pre-processing component

The log database has a typical relational form, where each record corresponds to a specific event. More specifically, each row in the log database contains the trace identifier, the event type, the timestamp and any other application-specific metadata that play no role in our generic solution. The second input of the pre-processing component contains the more recent log entries that have not been indexed yet. For example, if the index is updated on a daily basis, the log file is expected to contain from a few thousand of events up to several millions.

[3]https://cassandra.apache.org/

trace: <(A,1), (A,2), (B,3), (A,4), (B,5), (A,6)>		
Pair	**Strict Contiguity**	**Skip till next match**
(A,A)	(1,2)	(1,2),(4,6)
(B,A)	(3,4),(4,5)	(3,4),(5,6)
(B,B)	-	(3,5)
(A,B)	(2,3),(4,5)	(1,3),(4,5)

Table 3: Pairs created per different policy.

Pattern indexing and querying is applied per trace. In other words, for each distinct trace, a large sequence of all its events is constructed sorted by the event timestamps. To this end, the recent logfile is combined with the log database. In addition, and since the trace may span many indexing periods, new log entries need to combined with already indexed events in the same trace in a principled manner to avoid duplicates. If new logged events belong to a trace already started, we extract stored information from the indexing database (the exact procedure will be described in detail shortly).

Based on these trace sequences, we build an inverted indexing of all event pairs. That is, we extract all event pairs from each trace, and for each pair we keep its trace along with the corresponding pair of timestamps. This information is adequate to answer pattern queries efficiently, where these queries may not only refer to pattern detection, but frequency counts and prediction of next events, as discussed in Section 3.2.1. The index contains entries of the following format: `(A,B):{(trace12, 2,5),(trace12, 7,11), (trace15,1,6),. . . }`. In this example, the pair of event types `(A,B)` has appeared twice in trace12 at timestamps (2,5) and (7,11), respectively, and once in trace15.

The pre-processing component is implemented as a Spark Scala program to attain scalability. Next, we delve into more details regarding pre-processing subparts.

3.1.1 Creation of event pairs. There are more than one ways to create pair of events in a trace, which depends on the different policy applied. We have already given two policies, namely SC and STNM, which impact on how pairs are created.

Let us assume that a specific trace contains the following sequence of pairs of event types and their timestamps: `trace: <(A,1), (A,2), (B,3), (A,4), (B,5), (A,6)>`. Table 3 shows the pairs created per different policy. This example shows a simplified representation of the inverted indexing. SC detects only the pairs of events that are consecutive. There is no pair pair `(B,B)` because there is an event `(A)` between the two Bs in the trace. As expected, the SC policy creates less pairs per trace and is also easier to implement.

STNM skips events until it finds a matching event, but there are no overlapping pairs, the timestamps of which are intertwined. For example, regarding pair `(A,B)`, we consider only the (1,3) pair of timestamps and not (2,3). The complexity of pair creation in STNM is higher and there are several alternatives that are presented in Section 4.

A final note is that our approach can work even in the absence of timestamps. In that case, the position of an event in the sequence can play the role of the timestamp.

Algorithm 1 Update index

```
1: Input : new_events
2: traces ← transform new_events to traces as in the Seq table
3: temp ← LastChecked table joined with traces
4: new_pairs ← [ ]
5: for all trace in traces do
6:     extract events
7:     for all (ev_a, ev_b) do
8:         lt ← temp.get(ev_a, ev_b).last_completion for the same trace
9:         if ev_a.ts > lt then
10:            new_pairs += create_pairs(ev_a, ev_b)
11:        end if
12:    end for
13: end for
14: append new_pairs to the Index table
```

Algorithm 2 Pattern detection

```
1: procedure getCompletions(< ev_1, ev_2, ..., ev_p >)
2:     previous ← Index.get(ev_1, ev_2)
3:     for i =2 to p − 1 do
4:         idx_completions ← Index.get(ev_i, ev_{i+1})
5:         for all c in idx_completions grouped by trace do
6:             new ← [ ]
7:             for all pr in previous for the same trace do
8:                 if pr.last_event.ts == c.first.ts then
9:                     append c to pr and add to new
10:                end if
11:            end for
12:            previous ← new
13:        end for
14:    end for
15:    return previous
16: end procedure
```

3.1.2 Tables in the indexing database. The pre-processing phase creates and updates a set of tables, which can all be stored in a key-value store, such as Cassandra. The first one contains the trace sequence, so that there is no need to be reconstructed from scratch every time is needed, e.g., to append new events. The second one is the index presented earlier. The other tables are auxiliary ones, which are required during index creation and query answering.

- **Seq** with key: $trace_{id}$ and value: $\{(ev_a, ts_a), (ev_b, ts_b), ...\}$. This table contains all traces that are indexed. It is used to create and update the main index; new events belonging to the same trace are appended to the value list.
- **Index** with a complex key: (ev_a, ev_b) and value containing a list of triples: $\{(trace_{id}, ts_a, ts_b), ...\}$. This is the inverted index, which is the main structure used in query answering.
- **Count** with key a single event type: ev_a and value a list of triples: $\{(ev_b, sum\_duration, total\_completions), (ev_c, sum\_duration, total\_completions), ...\}$. For each event ev_a, we keep a list which contains the total duration of completions for a pair (ev_a, ev_x) and the total number of completions. This is used to find the most frequent pairs where an event appears first and also we can leverage the duration information in case further statistics are required.
- **Reverse Count**, which has exactly the same form of key and value with **Count**, but the statistics refer to pairs that have the event in the key as their second component
- **LastChecked** with complex key a pair (ev_a, ev_b) and value a list of pairs: $\{(trace_{id}, last\_completion), ...\}$. The length of the list is the number of traces in which the pair (ev_a, ev_b) appears. The $last\_completion$ field keeps the last timestamp of ev_b in a pair detection. This table is used to prevent creating and indexing pairs more than once.

3.1.3 Index update. In dynamic environments, new logs arrive continuously, but the index is not necessarily updated upon the arrival of each new log record. New log events are batched and the update procedure is called periodically, e.g., once every few hours. To avoid the generation of duplicates, the LastChecked table introduced above plays a crucial role. The index update rationale is illustrated in Algorithm 1.

In line 3 of the algorithm, we extract the LastChecked table and keep only its part that refers to the traces that their id appears in new events. In line 10, the *create_pairs* procedure is not specifically described here but can be any of the algorithms presented in Section 4 depending also on the policy employed.

A subtle point is that the index may grow very large. To mitigate this, a separate index table can be used for different periods, e.g., for different months. In addition, the traces corresponding to completed sessions can be safely pruned from the Seq table, along with the corresponding value entries in LastChecked.

3.2 The query processor component

The architecture described can support a range of pattern queries that are presented in Section 3.2.1. In Section 3.2.2, we give different solutions for predicting subsequent events in a pattern while trading off accuracy for response time.

3.2.1 Different type of queries. The query input is a pattern (i.e., a sequence) of events $< ev_1, ev_2, ev_3, \ldots, ev_p >$ for all supported query types. The query types in ascending order of complexity are as follows:

- **Statistics.** This type of query returns statistics regarding each pair of consecutive events in the pattern. The statistics are those supported by the Count table, namely number of completions and average duration. Also, from the LastChecked table, the timestamp of the last completion can be retrieved. The pairwise statistics can provide useful insights about the behavior of the complete pattern with simple post-processing and without requiring access to any other information. For example, the minimum number of completions of a pair provides an upper bound of the completions of the whole pattern in the query. Also, the sum of the average durations gives an estimate of the average duration of the whole pattern. Finally, the number of completions could be more accurately bounded if all pairs in the pattern are considered instead of the consecutive ones only; clearly, there is a tradeoff between result accuracy and query running time in this case.
- **Pattern Detection.** This query aims to return all traces that contain the given pattern. Query processing starts by searching for all the traces that contain event pair (ev_1, ev_2). At the next step, the technique keeps only the traces where the same instance of ev_2 is followed by e_3 to the pattern; to this end, it finds all the traces that contain (ev_2, ev_3) and keeps those for which ev_2 has the same timestamp in both cases. Up to now, we have found the traces that contain

Algorithm 3 Accurate exploration of events

1: **Input:** pattern $e_1, ev_2, ..., ev_p$
2: candidate_events ← from Count Table get all events that has ev_p as first event
3: propositions← [ ]
4: **for all** *ev* in candidate events **do**
5: tempPattern ← append *ev* to pattern
6: candidate_pairs ← **getCompletions**(TempPattern)
7: proposition ← apply time constraints to candidate pairs (optional)
8: append proposition to propositions
9: **end for**
10: return propositions sorted according to Equation (1)

(ev_1, ev_2, ev_3). The execution continues in the same way up to ev_p, as shown in Algorithm 2. It is trivial to extend the results with further information, such as the starting and ending timestamp.

- **Pattern Continuation.** Another aspect for pattern querying is exploring which events are most likely to extend the pattern in the query. This has several applications, such as predicting an upcoming event given partial pattern in an incomplete trace, or computing the probability of an event to appear in a pattern, based on prior knowledge. In this query, the response contains the most likely events that can be appended to the pattern, based on a scoring function. Equation 1 gives a score for a proposed event. Total completions refer to the frequency of this event to follow the last event in the query pattern, while average duration favors events that appear closer to the pattern in the original traces.

$$Score = \frac{total\_completions}{average\_duration} \quad (1)$$

3.2.2 Pattern Continuation Alternatives. Exploring events for pattern continuation can be computationally intensive, depending on the log size. Some times, we want accurate responses, while in other cases it is adequate to receive coarser insights so that we can trade accuracy for response time. We present three alternative ways of exploring events, namely one accurate, one fast heuristic and one hybrid that is in between the previous two.

- **Accurate.** In Algorithm 3, we present the outline of this method. In line 2 we use the Count table to find all event pairs that begin with the last event of the pattern and collect the second events of the pairs in the candidate_events list. The procedure getCompletions is already provided in Algorithm 2. We also allow for constraints in the average time between the last event in the pattern and the appended event; these constraints are checked in line 7. The strong point of this approach is that all pattern continuations are accurately checked one-by-one; the drawback is that the response time increases rapidly with the size of log files and the number of different events.

- **Fast.** In Algorithm 4 we perform a heuristic search. We start by finding the upper bound of the total times the given pattern has been completed (lines 3-8). Then, for every possible event *ev*, we approximate the upper bound if this event is added at the end of the pattern, by keeping the minimum between the max_completions and the total completions of *ev* (line 11). The strong point of this approach is that it is fast, since it extracts precomputed statistics from the indexing database but the results rely only on approximations.

Algorithm 4 Fast exploration of events

1: **Input:** Pattern $e_1, ev_2, ..., ev_p$
2: max_completions ← ∞
3: **for all** (ev_i, ev_{i+1}) in pattern **do**
4: count ← Count Table get ev_i, ev_{i+1}
5: **if** count.total_completions < max_completions **then**
6: max_completions ← count.total_completions
7: **end if**
8: **end for**
9: propositions ← [ ]
10: **for all** ev in Count.get(ev_p) **do**
11: completions ← min(max_completions,ev.total_completions)
12: append (ev.event,completions,ev.average_duration) to propositions
13: **end for**
14: return propositions sorted according to Equation (1)

Algorithm 5 Hybrid exploration of events

1: **Input:** Pattern $e_1, ev_2, ..., ev_n$
2: **Input:** topK
3: fast_propositions ← run Algorithm 4 for the input pattern
4: propositions ← run Algorithm 3 for topK of fast_propositions
5: return propositions sorted according to Equation (1)

- **Hybrid.** Lastly, in Algorithm 5 we perform a trade off between accuracy and response time. This flavor receives topK as an input parameter. First, the fast alternative runs to provide an initial ranking of possible pattern continuations. Then, for the topK intermediate results, the accurate method runs. In this flavor, the trade-off is configurable. Setting topK to l, the technique degenerates to the accurate, while setting topK to 0 is equal to the fast only alternative.

4 ALTERNATIVES FOR INDEXING EVENT PAIRS

The indexing of event pairs largely depends on the pattern detection policy. For the Strict contiguity (SC) policy, the process is straightforward. For Skip-till-next-match (STNM), there are three flavors. Each trace is processed separately in parallel using Spark. Below, we show the processing per trace; therefore the overall complexity for the complete log needs to be increased by a factor of O(m). The techniques presented in this section refer to the implementation of the *create_pairs* procedure in Alg. 1.

4.1 Strict Contiguity

This method is straightforward: we parse each trace and we add the consecutive trace events in the index. The complexity is $O(n)$, where n is the size of a trace sequence in the log file.

4.2 Skip-till-next-match

The three different ways to calculate the event pairs using the skip-till-next-match (STNM) strategy have different perspectives.

Algorithm 6 Parsing method (per trace)

```
    checkedList ← [ ]
    for i in (0, trace.size-1) do
        inter_events ← [ ]
        if ev_i.type not in checkedList then
5:          for j in (i,trace.size) do
                if ev_i.type==ev_j.type then
                    update inv_index with (ev_i, ev_j)
                    for all inter_e in inter_events do
                        update inv_index with (ev_i, inter_e)
10:                 end for
                    reset inter_events
                else if ev_j.type not in inter_events then
                    update inv_index with ev_i, ev_j
                    append ev_j to inter_events
15:             end if
            end for
            append ev_i to checkedList
        end if
    end for
```

The *Parsing* method computes pairs while parsing through the sequence. The *Indexing* method, first detects the positions of each distinct event and then calculates the pairs. Finally, the *State* method updates and saves a state for the sequence for each new event.

Each method can be used in different scenarios. As we will show in the experimental section, the *Indexing* method dominates in the settings investigated. But this is not necessarily always the case. For example, if we operate in a fully dynamic environment, where new events are appended continuously as a data stream, its easier to keep a state of the sequence than calculating all the pairs from the start. However, in our core scenario where new logs are processed periodically, all three ways apply. In addition, if a domain has a lot of distinct events, i.e., l is very high and much higher than the cardinalities examined, the *Indexing* method becomes inefficient and thus is better to use the *Parsing* one.

Parsing method. The main structure is inv_index, which is a trace-specific part of the Index table. For each trace, the entries of this table are augmented in parallel and since there is no ordering in the values, this is not a problem.

The main rationale is shown in Algorithm 6, which contains two main loops, in line 2 and in line 5, respectively. The idea is to create all the event pars (ev_i, ev_j), in which ev_i is before ev_j. The checkedList prevents the algorithm from dealing with events types that has already been tested. While looping through the trace sequence for an event ev_1, the algorithm appends all new events to inter_events until it finds an event, ev_2 that has the same type as ev_1. When this happens it will create all the pairs of ev_1 with the events in the inter_events list (line 8-10) and will empty it (line 11). After that point, the algorithm proceeds with creating pairs where the timestamp of the first event is now equal to ev_2's timestamp. While updating the index, some extra checks are performed to prevent entering the same pairs twice.

Complexity Analysis. Even though there are two loops iterating the events, the if statement in line 4 can be true only up to l times (where l is the number of distinct elements) and so the complexity is $O(nl^2)$, with n being the length of the trace sequence. The space required is $O(n+l^2)$, for the inv_index and the checkedList.

Algorithm 7 Indexing method (per trace)

```
1:  indexer ← map(event_id):[timestmap1,timestamp2,...]
2:  for all ev_a in indexer do
3:      for all ev_a in indexer do
4:          CreatePairs(indexer[ev_a.tsList],indexer[ev_b.tsList]))
5:      end for
6:  end for
    procedure CreatePairs(times_a,times_b)
        i,j,prev ← 0,0,-1
        pairs ← [ ]
        while i < times_a.size and j < times_b.size do
5:          if times_a[i] < times_b[j] then
                if times_a[i] > prev then
                    append (times_a[i],times_b[j]) to pairs
                    prev ←times_b[j], i←1, j←1
                else
10:                 i←1
                end if
            else
                j←1
            end if
15:     end while
        return pairs
    end procedure
```

Algorithm 8 State method (per trace)

```
1:  index ← HashMap((ev_i, ev_j):[ts_1, ts_2, ts_3...])
2:  for all ev in the trace do
3:      Add_New(index,ev)
4:  end for
5:  return index
6:  procedure Add_New(index, new_event)
7:      for all combinations where new_event is the 1st event
    in index do
8:          update state
9:      end for
10:     for all combinations where new_event is the 2nd event
    in index do
11:         update state
12:     end for
13: end procedure
```

Indexing method. The key idea is to read the whole sequence of events while keeping the timestamps in which every event type occurred (line 1). Then, for every possible combination of events we run the procedure, in which we create the pairs. The procedure is similar to a merging of two lists, while checking for time constrains. More specifically, in line 5, the order of the events is checked and then in line 6, we ensure that there are no overlapping event pairs.

Complexity Analysis. In line 1, we loop once the entire sequence to find the indexes of each distinct event ($O(n)$). Then, the next loops in lines 2-3 retrieve all the possible event pairs ($O(l^2)$) and finally the procedure in line 4, will pass through their indices ($O(n)$). This gives a total complexity of $O(n+l^2n)$, which is simplified to $O(nl^2)$. The total space required is $O(n+l^2)$, for the partial and the pairs. I.e., the complexity is similar to the *Parsing* method.

State method. The algorithm is based on a Hash Map, which contains a list of timestamps for each pair of events. We first initialize the structure by adding all the possible event pairs that can be created (line 1) through parsing the sequence and detecting all distinct event types that appear. Then we loop again through the event sequence. While looping through the sequence we add new event (ev_i) in the structure, by first updating all the the pairs that have ev_i as the first event and then as the second (procedure lines 7-12). During these updates, we ensure that no pair is overlapping. The update operation is as follows. For each (ev_i, ev_j) entry in the HashMap, if the list of the timestamps has even size, we append $ev_i.ts$; otherwise we do nothing. Similarly, for each (ev_j, ev_i) entry in the HashMap, if the list of the timestamps has odd size, we append $ev_i.ts$; otherwise we do nothing. At the end, we trim all timestamp lists of odd size (not shown in the algorithm).

Complexity Analysis. The space complexity is $O(l^2)$ due to the HashMap. In line 2, the loop is passing through all the events in the sequence and for every event executes the procedure Add_new. This procedure has two loops passing through the set of distinct events (l), which gives us a total complexity of $O(nl)$ multiplied by the complexity to access the HashMap, which is $O(1)$ in the average case. Despite this lower complexity, in the evaluation section, we will provide evidence that the overheads due to the HashMap access are relatively high.

Implementation information. We have used Spark and Scala for developing the pre-processing component, which encapsulates the event pair indexing, and Java Spring for the query processor. The source code is publicly available on GitHub.[4],[5]

5 EVALUATION

We used both real-world and synthetic datasets to evaluate the performance of the proposed methods. We start by presenting the datasets, followed by the evaluation of the different flavors of indexing event pairs. Then we compare the preprocess time with the proposal in [19] and Elasticsearch v7.9.1 and finally we show the response time for queries that executed in both methods. In query processing, we also compare against SASE [34].[6] All tests were conducted on a machine with 16GB of RAM and 3.2GHz CPU with 12 cores. Cassandra is deployed on a separate machine with 64GB of RAM and 2GHz CPU. Each experiment is repeated 5 times and the average time is presented.

5.1 Datasets

The real-world datasets are taken from the Business Process Intelligence (BPI) Challenges, and more specifically from the years 2013, 2017 and 2020. BPI13[7] is an event log of Volvo IT incident and problem management. It includes 7,554 traces, which contain 65,533 events in total. The mean, min and max number of events per trace for this dataset are 8.6, 1 and 123, respectively. BPI17[8] is an event log, which corresponds to a loan application of an Dutch financial institute. It includes 31,509 traces, which contain over 1M (1,202,267) events in total. The mean, min and max number of events per trace for this dataset are 38.15, 10 and 180, respectively.

[4]https://github.com/mavroudo/SequenceDetectionPreprocess
[5]https://github.com/mavroudo/SequenceDetectionQueryExecutor
[6]The SASE code repository used in the experiments is https://github.com/haopeng/sase
[7]doi:10.4121/500573e6-accc-4b0c-9576-aa5468b10cee
[8]https://data.4tu.nl/articles/BPI_Challenge_2017/12696884

Log file	Number of traces	Activities
max_100	100	150
max_500	500	159
med_5000	5,000	95
max_5000	5,000	160
max_1000	1,000	160
max_10000	10,000	160
min_10000	10,000	15
bpi_2013	7,554	4
bpi_2020	6,886	19
bpi_2017	31,509	26

Table 4: Number of traces and distinct activities for every process-like event log.

From BPI20[9], we use an event log of requesting for payment for a business trip. This is the smaller real-world dataset. It includes 6,886 traces, which contain 36,796 events. The mean, min and max number of events per trace for this dataset are 5.3, 1 and 20, respectively.

We also created synthetic datasets. First with the help of the PLG2[10] tool, we created 3 different processes, with different number of distinct activities (15,95,160). Then by modifying the number of traces per logfile, we created logs that contain from 500 to 400,000 events. The log files are in the XES[11] format. In Figure 2, the distributions of events per trace and unique activities per trace are shown. The purpose of these figures is to provide evidence that our test datasets cover a broad range of real-world trace profiles, thus the experimental results are trustworthy. In general, logs with the terms "med" and "max" in their name have more events per trace and much more unique activities than those with the term "min". Summary metadata are also in the Table 4. The process-oriented logs are not big, but are used in order to compare our approach against the one in [19], which has been employed in pattern continuation in business processes. This method cannot handle much bigger datasets. To test the scalability of our solution, we employ some additional random datasets that will be introduced separately.

5.2 Evaluating the different ways of indexing pairs

In this section, we evaluate the different flavors that index the event pairs according to the skip-till-next-match (STNM) policy. We aim to find the pros and cons for each flavor in Section 4 and also define the different real life scenarios to use them complementing the discussions already made above. We start the evaluation using the datasets in Table 4. The results are shown in Table 5. The main observation is that all three flavors perform similarly while indexing process-like datasets. When the relative differences are larger (e.g., larger than 30% for bpi_2020), the absolute times are small, so the impact of different decisions is not that important.

These datasets are not big. To better test the potential of the three alternatives, we created log files in which the events were not based on a process. We range the number of traces from 100 to 5000, the number of max events per trace from 50 to 4000

[9]https://doi.org/10.4121/uuid:52fb97d4-4588-43c9-9d04-3604d4613b51
[10]https://plg.processmining.it/
[11]https://xes-standard.org/

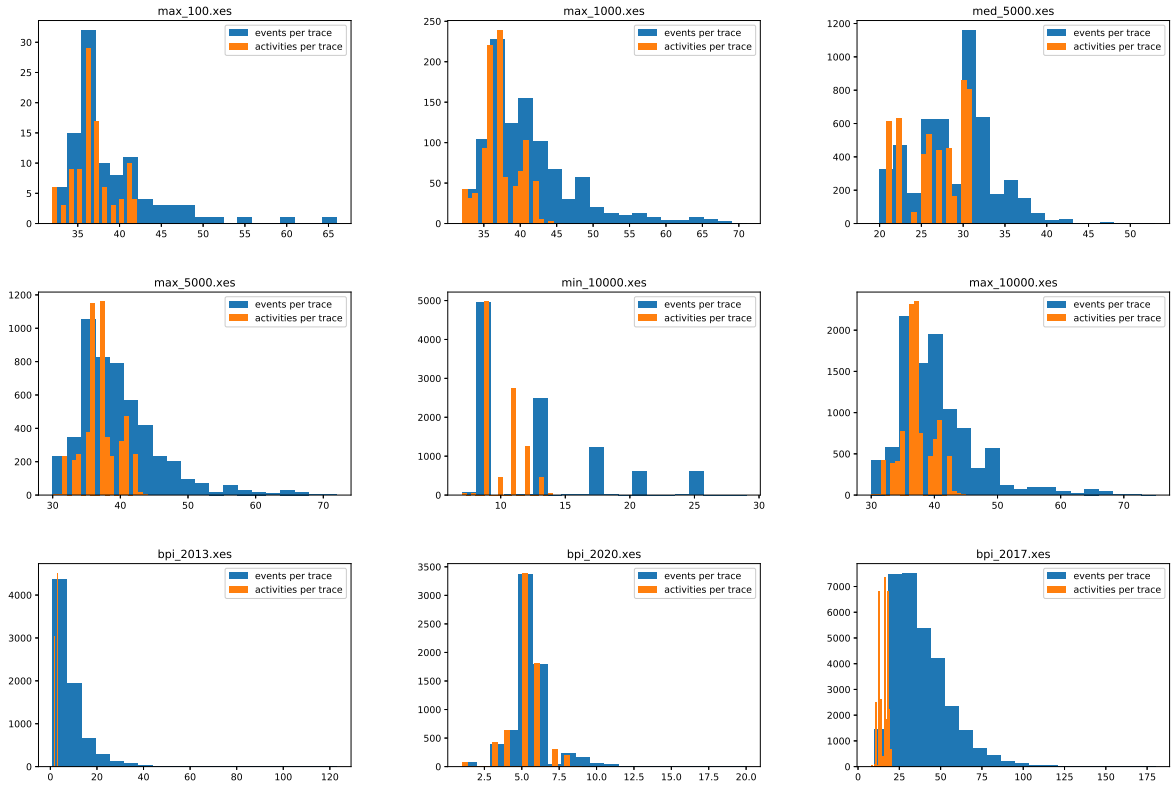


Figure 2: Distributions of the number of events and activities (i.e., unique event types) per trace for every process like log file.

Log file	Indexing	Parsing	State
max_100	4.874	4.49	4.572
max_500	8.454	7.109	7.294
max_1000	10.656	10.407	10.447
med_5000	23.105	22.601	22.417
max_5000	38.152	34.854	38.444
max_10000	79.863	77.964	80.796
min_10000	15.604	13.979	13.625
bpi_2020	6.803	10.384	8.822
bpi_2013	9.528	8.044	8.197
bpi_2017	170.9	171.666	179.352

Table 5: Execution times of different methods (in seconds).

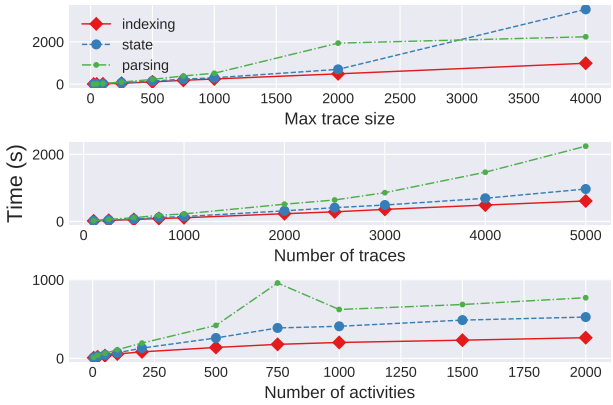


Figure 3: Comparison of execution times of the three different approaches of indexing the event pairs according to the STNM policy for large random logs.

and the number of activities from 4 to 2000. We refer to them as random datasets, due to the lack of correlation between the appearance of two events in a trace, which is not the typical case in practice, and renders the indexing problem more challenging.

The results are presented in Figure 3. In the first plot, we set the number of traces equal to 1000 and the number of different activities equal to 500, while changing the number of max events per trace from 100 to 4000. I.e., we handle up to 4M events. In the second plot, we keep the maximum number of events per trace and distinct activities to 1000 and 100, respectively while increasing the number of traces from 100 to 5000. I.e., we handle up to 5M events. Lastly, we maintain both the number of traces and maximum number of events to 500 and increase the distinct activities from 4 to 2000.

From the Figure 3, we can observe that the Indexing alternative outperforms the other two, in some cases by more than an order of magnitude. The simplicity of this method makes it superior to State, even though the time complexity indicates that the latter is better. The State method performs better than Parsing; especially in the third plot we can see the non-linear correlation between the execution time and the number of distinct activities.

In summary, our results indicate that indexing is the most efficient flavor to use when dealing with log files considered periodically (so that new log entries are a few millions): it has

Log file	[19]	Strict (1 thread)	Strict	Indexing (1 thread)	Indexing	Elasticsearch
max_100	1.054	3.764	3.701	5.398	4.874	0.67
max_500	2.68	5.593	4.649	12.568	8.454	4.68
max_1000	4.458	7.084	5.69	22.544	10.656	10.167
med_5000	6.913	20.361	9.175	113.04	23.105	31.80
max_5000	16.163	25.419	12.452	210.713	38.152	31.41
min_10000	26.64	31.379	8.782	116.318	15.604	38.15
max_10000	37.569	63.975	21.006	734.844	79.863	121.167
bpi_2020	95.269	11.461	8.597	17.908	6.803	14.49
bpi_2013	504.089	12.817	7.918	14.925	9.528	9.973
bpi_2017	very high	451.666	66.284	crash	170.9	364.293

Table 6: Comparison of execution times between [19] and our proposal (time in seconds).

minimum space complexity and it has the best executing time. On the contrary, State is preferable when operating in a dynamic environment, when for example new logs will be appended at the end of every few minutes and some traces will be active for weeks. State allows to save the current state of the log and, when new events are appended, it can calculate the event pairs without checking the previous ones. Even though the space complexity is higher than the other methods, it is expected to dominate in a real dynamic scenario.

5.3 Pre-process comparison

Based on the previous results, we continue the comparison using only the Indexing alternative for the STNM policy. We compare the time for building the index for both SC and STNM against [19], which supports only SC, and against Elasticsearch. The results are presented in Table 6; to provide the full picture we run Spark in two modes, namely using all the available machine cores and using a single Spark executor. The latter allows for direct comparison against non-parallel solutions.

Considering how [19] works, logs that are based on processes are easier to handle. We can split the test datasets of table 4 into three categories, namely small synthetic datasets (100-1000 traces), large synthetic datasets (5000 & 10000 traces) and real datasets (from the BPI challenge). In the first category, Strict performs almost the same as [19], while Indexing has significantly higher execution time, due to the more complex process it runs. In the second category, Strict scales better and achieves better times than [19]. Finally, in real datasets, our method achieves two order of magnitude lower times compared to [19]. When using the bpi_2017 dataset, [19] could not even finish indexing in 5 hours. This is probably based on the large amount of events (≈1.2M) combined with the high number of distinct events per trace. This lead to a very large suffix array, which probably could not fit in main memory and ended up doing an extensive amount of I/Os. For the same dataset, both Indexing and Strict managed to create inverted indexing in less than 3 minutes when using all machine cores.

Compared to Elasticsearch, we can observe that our best performing technique is on average faster for the last two categories (large synthetic and real datasets). In the larger real dataset, building an index to support STNM queries according to our proposal is more than 2.1X faster than Elasticsearch.

Parallelization-by-design is a big advantage of our method; we do not simply employ Spark but we can treat each trace in parallel. Further, parallelization applies to both the event-pair creation and the storage (Cassandra is a distributed database). As shown in Table 6, indexing can run even 10 times faster when using all 12 cores available. This is not the case for the [19] and other solutions, like Elasticsearch. However, there exist some structures that build suffix trees in parallel [2, 13, 20]. But still, the most computational intense process is to find all the subtrees and store them. The number of subtrees is increased with the number of leaves, which depends on the different traces that can be found in a logfile.

Log file	[19]	Our method (2)	Our method (10)
max_100	0.0023	0.007	0.022
max_500	0.0026	0.020	0.029
max_1000	0.0022	0.010	0.050
med_5000	0.0022	0.013	0.280
max_5000	0.0026	0.007	0.230
min_10000	0.0022	0.060	2.200
max_10000	0.0026	0.012	0.400
bpi_2020	0.0059	0.006	0.290
bpi_2013	0.0185	0.034	4.000

Table 7: Comparison response times in seconds

5.4 Query response time

We start by comparing the response time for a single query, between our method and the one in [19]. Since [19] supports the strict contiguity (SC) policy solely, we use this policy to create the inverted index and then execute a pattern detection query, as described in Section 3.2.1. Then, we compare the STNM solutions against Elasticsearch and SASE, which does not perform any preprocessing. We do not employ Elasticsearch in the SC experiments, because it is more suitable for STNM queries; more specifically, supporting SC can be achieved with additional expensive post-processing. Finally, we compare the pattern continuation methods and show the effectiveness of the trade-off between accuracy and response time.

5.4.1 Comparison against [19] for SC. The results of the comparison are shown in Table 7. In the first column, we can see the response time of [19] for the different log files. In the next 2 columns, we have different response times for detection queries, for pattern length equal to 2 and 10, respectively. As discussed in Section 2.2, all subtrees are precalculated and stored, which means that the detection query time is $O(logn + k)$ where n here is the number of different subtrees and k is the number of subtrees that will return. As such, for [19], the response time does not depend on the querying pattern length. On the other hand,

Log file	Elasticsearch	SASE	Our method
pattern length = 2			
max_100	0.006	0.003	0.003
max_500	0.009	0.014	0.006
max_1000	0.009	0.038	0.004
med_5000	0.048	0.958	0.006
max_5000	0.015	1.400	0.005
min_10000	0.145	1.565	0.031
max_10000	0.048	7.024	0.011
bpi_2013	0.071	0.205	0.008
bpi_2020	0.068	0.366	0.040
bpi_2017	0.609	70.491	0.102
pattern length = 5			
max_100	0.011	0.002	0.008
max_ 500	0.018	0.014	0.012
max_ 1000	0.017	0.038	0.013
med_ 5000	0.126	0.999	0.048
max_ 5000	0.037	1.226	0.036
min_ 10000	0.647	1.688	0.525
max_ 10000	0.170	6.413	0.061
bpi_ 2013	0.155	0.233	0.063
bpi_ 2020	0.246	0.534	0.562
bpi_ 2017	4.652	370.142	1.495
pattern length = 10			
max_100	0.020	0.002	0.048
max_500	0.031	0.014	0.039
max_1000	0.032	0.038	0.060
med_5000	0.239	1.010	0.279
max_ 5000	0.075	1.245	0.218
min_10000	1.340	1.712	3.707
max_10000	0.289	6.491	0.373
bpi_2013	0.259	0.229	0.374
bpi_2020	0.440	0.531	4.262
bpi_ 2017	9.661	440.066	11.188

Table 8: Response times for STNM queries in seconds

our proposal incrementally calculates the completion for every event in the pattern as described in Section 3.2.1; as such, the response time depends on the pattern length. In Figure 4, we show how response time increases with respect to the querying pattern length.

The experiments in Table 7 were executed 5 times and we presented the mean response time. However, we have noticed a fluctuation in response times, which is affected by the events in the pattern. Each event has a different frequency in the log files; e.g., starting and ending events are more frequent than some events that correspond to an error. When events in the querying pattern have low frequency, the response time will be shorter because there are fewer traces that need to be tested.

For small patterns, with length equal to 2-5, we get similar response times between the two methods, while [19] is always faster. As pattern length increases, our method's response time increases as well, but we also return as a by-product detection for all the sub-patterns.

In summary, the table shows the penalty we pay against a state-of-the-art technique during subsequence matching; however, the benefits of our approach are more significant: we allow efficient indexing in large datasets and we support, with similar times, pattern queries using the STNM policy.

5.4.2 Comparison against Elasticsearch and SASE for STNM. In Table 8, we present comparison against SASE and Elasticsearch

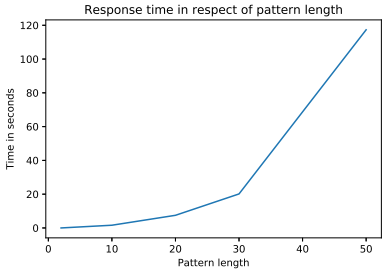


Figure 4: Response time with respect to the querying pattern length

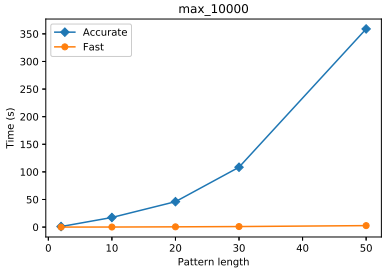


Figure 5: Response time for different pattern continuation methods for different query pattern lengths

query response times, when, in each experiment, we search for 100 random patterns. There are two main observations. Firstly, running techniques that perform all the processing on the fly without any preprocessing, such as SASE, yields acceptable performance in small datasets but significantly degrades in larger datasets, such as bpi_2017 and max_10000. In the former dataset, techniques that perform preprocessing are faster by 2 orders of magnitude. Secondly, there is no clear winner between Elasticsearch and our solution. But, in general, we are faster for small queries of pattern size equal to 2 and in all but one dataset for pattern size equal to 5, while Elasticsearch runs faster for pattern length equal to 10. However, for the longest running long queries, our solution is only 15.8% slower. Therefore, we can claim that our solution is competitive for large query patterns. Moreover, we can relax our query method to achieve faster times, as explained in the next part of the evaluation, while we support pattern continuation more efficiently due to the incremental approach of pattern processing that we adopt; i.e., we do not have to repeat the complete query from scratch.

5.4.3 Comparison of pattern continuation alternatives. In Figure 5, we show the response times between Accurate and Fast method for the dataset max_10000. We can see that the Accurate method follows the same pattern as the graph in Figure 4, which is what we expected as it performs pattern detection for every possible subsequent event in the pattern. On the other hand, there is no significant increase of response time for the Fast heuristic with regards of the pattern length.

We are trying to fill this performance gap with the Hybrid alternative. In Figure 6, we use again the max_10000 dataset and a pattern with 4 events and we show the response time with respect to the *topK* parameter given to Hybrid. The response time for both Accurate and Fast is constant, because they do not use this parameter. As expected, the time increases linearly as k increases.

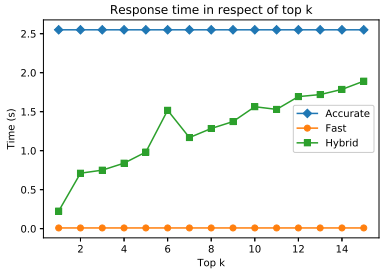


Figure 6: Response time of pattern continuation methods for different *topK* values

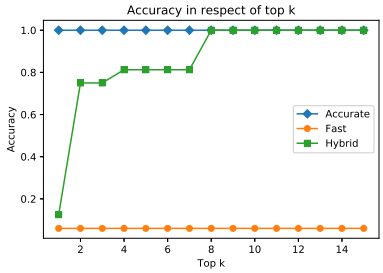


Figure 7: Accuracy of pattern continuation methods for different *topK* values

Fast's execution time is the lower bound and Accurate's is the upper one.

For the same setup, we perform an accuracy test presented in Figure 7. We use as ground truth the events returned from the Accurate method and compute the accuracy as the fraction of the events in the top k propositions from Hybrid that exist in the propositions reported by Accurate, where k is the number of propositions returned from Accurate. The accuracy is increasing as the number of k increases until it reaches 100% for k=8. For the same value, response time is half of the Accurate, as shown in the previous graph. Also in this example we could achieve a 80% accuracy with k=2 and 1/3 of the response time that Accurate would have taken.

6 RELATED WORK

Our work relates to several areas that are briefly described here in turn.

Complex event processing. There are number of surveys presenting scalable solutions for the detection of complex events in data stream. A variety of general purpose CEP languages have been developed. Initially, SASE [30] was proposed for executing complex event queries over event streams supporting SC only. The SQL-TS [24] is an extension to the traditional SQL that supports search for complex and recurring patterns (with the use of Kleene closure), along with optimizations, to deal with time series. In [9], the SASE language was extended to support Kleene closures, which allow irrelevant events in between thus covering the skip-till-next-match (STNM) and skip-till-any-match strategies. An extensive evaluation of the different languages was presented in [34] along with the main bottlenecks for CEP. In addition, the K*SQL [21] language, is a strict super-set of SASE+, as it can work with nested words (XML data). Besides the languages, most of these techniques use automata in order to detect specific patterns in a stream, like [18]. Our technique differs from them as we do not aim to detect patterns on the fly, but instead, to construct the infrastructure that allows for fast pattern queries in potentially large databases. To this end, the work in [22] also uses pair of events to create signatures, but for scalability purposes, this work considers only the top-k most frequent pairs, which yields an approximate solution, whereas we focus on exact answers. In addition, [22] focuses on the proposal of specific index types, whereas we follow a more practical approach, where are indices are stored as Cassandra tables to attain scalability.

Pattern mining. For non-streaming data, a series of methods have been developed in order to mine patterns. The majority of these proposals are looking for frequent patterns; e.g., Sahli et al in [26] proposed a scalable method for detecting frequent patterns in long sequences. As another example, in several other fields such as biology, several methods have developed, which are typically based in statistics (e.g., [1, 15]) and suffix trees (e.g., [10]). Parallel flavors have also been proposed, e.g., in [7]. Other forms of mined patterns include outlying patterns [5] or general patterns with high utility as shown in [32]. It is not trivial to build on top of these techniques to detect arbitrary patterns, because these techniques typically prune non-interesting patterns very aggressively.

Business processes. There are applications in business process management that employ pattern mining techniques to find outlier patterns and clear log files from infrequent behavior, e.g., [16, 28], in order to facilitate better process discovery. Another application is to predict if a trace will fail to execute properly; for example, in [4, 17], different approaches to predicting the next tasks of an active trace are presented. None of these techniques addresses the problem of efficiently detecting arbitrary sequences of elements in a large process database as we do, but the technique in [27] encapsulates our main competitor, namely [19]. Finally, in [8], a query language over business processes was presented to support sub-process detection. The proposed framework can be leveraged to support SC and STNM queries over log entries rather than subprocesses, but this entails using a technique like SASE, without any pre-processing. Our evaluation shows that such techniques are inferior to our solution.

Other data management proposals. The closest work to ours from the field of data management is set containment join queries, e.g., [31]. However, this type of joins does not consider time ordering. An interesting direction for future work is to extend these proposals to work for ordered sequences rather than sets; to date, this remains an open issue.

7 DISCUSSION

We have proposed a methodology to detect pattern according to the Strict contiguity (SC) and Skip-till-next-match (STNM) policy in large log databases, assuming that new events arrive and processed in big batches. However, there are several issues that need to be addressed with a view to yielding a more complete solution. First, sequential patterns, in their more relaxed form, allow for overlappings, which is commonly referred to as the skip-till-any-match policy. Supporting such patterns places additional burden to both the indexing process and query execution. Second, in many cases, assuming a total ordering is restrictive and also, the way some events may be logged, even in the same trace, cannot be regarded as following a total order. For example, in predictive maintenance in Industry 4.0, it is common to group events in large sets ignoring their relative order, e.g., [29]. Extending

our approach to operate under partial ordering is an interesting extension. Additionally, judiciously choosing the optimal update period is an open issue and gives rise to a multi-objective problem where low indexing time and result timeliness are contradicting objectives. Finally, the pattern continuation techniques can account for other operation modes, where an event is not appended only at the end, but also at arbitrary places in the query pattern. Our proposal can be easily extended to cover these cases, but we omit details here.

8 CONCLUSION

Despite the big advances in complex event processing and sequential pattern mining, efficient detection of arbitrary subsequences in log databases is an overlooked issue. Our proposal fills this gap and proposes indexing techniques along with query evaluation algorithms that allow the user to detect any patterns according to either the strict contiguity and the skip-till-next-match policy. Compared to subsequence matching techniques that support only strict contiguity, we show that our indexing can scale and also, query processing times are competitive when both approaches are applicable. Compared to Elasticsearch, a state-of-the-art solution, we build the indices faster and we run small queries faster, while we are competitive in large queries. Further, our solution can support exploration of pattern extension alternatives with different trade-offs between running time and accuracy and builds on top of scalable technologies, like Spark and Cassandra.

Acknowledgment. The research work was supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant" (Project Number:1052).

REFERENCES

[1] Alberto Apostolico, Matteo Comin, and Laxmi Parida. 2011. VARUN: Discovering Extensible Motifs under Saturation Constraints. *IEEE/ACM transactions on computational biology and bioinformatics / IEEE, ACM* 7 (01 2011), 752–26. https://doi.org/10.1109/TCBB.2008.123

[2] A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and Uzi Vishkin. 1988. Parallel construction of a suffix tree with applications. *Algorithmica* 3 (11 1988), 347–365. https://doi.org/10.1007/BF01762122

[3] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. 2010. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD Conference*. 975–986.

[4] Michael Borkowski, Walid Fdhila, Matteo Nardelli, Stefanie Rinderle-Ma, and Stefan Schulte. 2019. Event-based failure prediction in distributed business processes. *Information Systems* 81 (2019), 220 – 235. https://doi.org/10.1016/j.is.2017.12.005

[5] Lei Cao, Yizhou Yan, Samuel Madden, Elke A. Rundensteiner, and Mathan Gopalsamy. 2019. Efficient Discovery of Sequence Outlier Patterns. *Proc. VLDB Endow.* 12, 8 (April 2019), 920-932. https://doi.org/10.14778/3324301.3324308

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.

[7] Alexandra Carvalho, Arlindo Oliveira, Ana Teresa Freitas, and Marie-France Sagot. 2004. A Parallel Algorithm for the Extraction of Structured Motifs. *Proceedings of the ACM Symposium on Applied Computing* 1, 147–153. https://doi.org/10.1145/967900.967932

[8] Daniel Deutch and Tova Milo. 2012. A structural/temporal query language for Business Processes. *J. Comput. System Sci.* 78, 2 (2012), 583 – 609. https://doi.org/10.1016/j.jcss.2011.09.004 Games in Verification.

[9] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. 2007. *Sase+: An agile language for kleene closure over event streams.* Technical Report. UMass Technical Report.

[10] Avrilia Floratou, Sandeep Tata, and Jignesh Patel. 2010. Efficient and Accurate Discovery of Patterns in Sequence Datasets. *Proceedings - International Conference on Data Engineering*, 461–472. https://doi.org/10.1109/ICDE.2010.5447843

[11] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, and Yun Sing Koh. 2017. A Survey of Sequential Pattern Mining. *Data Science and Pattern Recognition* 1, 1 (2017), 54–77.

[12] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Tin Truong-Chi, and Roger Nkambou. 2019. A survey of high utility itemset mining. In *High-Utility Pattern Mining*. Springer, 1–45.

[13] Amol Ghoting and Konstantin Makarychev. 2009. Serial and Parallel Methods for i/o Efficient Suffix Tree Construction. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1559845.1559931

[14] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Complex event recognition in the Big Data era: a survey. *VLDB J.* 29, 1 (2020), 313–352.

[15] Roberto Grossi, Andrea Pietracaprina, Nadia Pisanti, Geppino Pucci, Eli Upfal, and Fabio Vandin. 2009. MADMX: A Novel Strategy for Maximal Dense Motif Extraction. https://doi.org/10.1007/978-3-642-04241-6_30

[16] Ying Huang, Yingxu Wang, and Yiwang Huang. 2018. Filtering Out Infrequent Events by Expectation from Business Process Event Logs. 374–377. https://doi.org/10.1109/CIS2018.2018.00089

[17] Bokyoung Kang, Dongsoo Kim, and Suk-Ho Kang. 2012. Real-time business process monitoring method for prediction of abnormal termination using KNNI-based LOF prediction. *Expert Systems with Applications* 39, 5 (2012), 6061 – 6068. https://doi.org/10.1016/j.eswa.2011.12.007

[18] Ilya Kolchinsky and Assaf Schuster. 2019. Real-Time Multi-Pattern Detection over Event Streams. 589–606. https://doi.org/10.1145/3299869.3319869

[19] Fabrizio Luccio, Antonio Mesa Enriquez, Pablo Olivares Rieumont, and Linda Pagli. 2001. *Exact Rooted Subtree Matching in Sublinear Time.* Technical Report. TR-01-14, University of Pisa.

[20] Essam Mansour, Amin Allam, Spiros Skiadopoulos, and Panos Kalnis. 2011. ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings. *CoRR* abs/1109.6884 (2011). arXiv:1109.6884 http://arxiv.org/abs/1109.6884

[21] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. 2010. From Regular Expressions to Nested Words: Unifying Languages and Query Execution for Relational and XML Sequences. *PVLDB* 3 (09 2010), 150–161.

[22] Alexandros Nanopoulos, Yannis Manolopoulos, Maciej Zakrzewicz, and Tadeusz Morzy. 2002. Indexing web access-logs for pattern queries. In *Fourth ACM CIKM International Workshop on Web Information and Data Management (WIDM 2002).* 63–68.

[23] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2004. Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach. *IEEE Trans. Knowl. Data Eng.* 16, 11 (2004), 1424–1440.

[24] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. 2001. Optimization of Sequence Queries in Database Systems. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '01).* Association for Computing Machinery, New York, NY, USA, 71-81. https://doi.org/10.1145/375551.375563

[25] Majed Sahli, Essam Mansour, and Panos Kalnis. 2014. ACME: A scalable parallel system for extracting frequent patterns from a very long sequence. *VLDB J.* 23, 6 (2014), 871–893.

[26] Majed Sahli, Essam Mansour, and Panos Kalnis. 2014. ACME: A scalable parallel system for extracting frequent patterns from a very long sequence. *The VLDB Journal* 23 (12 2014). https://doi.org/10.1007/s00778-014-0370-1

[27] Suhrid Satyal, Ingo Weber, Hye young Paik, Claudio Di Ciccio, and Jan Mendling. 2019. Business process improvement with the AB-BPM methodology. *Information Systems* 84 (2019), 283 – 298. https://doi.org/10.1016/j.is.2018.06.007

[28] Sebastiaan J. van Zelst, Mohammadreza Fani Sani, Alireza Ostovar, Raffaele Conforti, and Marcello La Rosa. 2020. Detection and removal of infrequent behavior from event streams of business processes. *Information Systems* 90 (2020), 101451. https://doi.org/10.1016/j.is.2019.101451 Advances in Information Systems Engineering Best Papers of CAiSE 2018.

[29] J. Wang, C. Li, S. Han, S. Sarkar, and X. Zhou. 2017. Predictive maintenance based on event-log analysis: A case study. *IBM Journal of Research and Development* 61, 1 (2017), 11.

[30] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. *Proceedings of the ACM SIGMOD International Conference on Management of Data* 10, 407–418. https://doi.org/10.1145/1142473.1142520

[31] Jianye Yang, Wenjie Zhang, Shiyu Yang, Ying Zhang, and Xuemin Lin. 2017. TT-Join: Efficient Set Containment Join. In *33rd IEEE International Conference on Data Engineering, ICDE.* 509–520.

[32] Junfu Yin, Zhigang Zheng, and Longbing Cao. 2012. USpan: An efficient algorithm for mining high utility sequential patterns. *KDD 2012* (08 2012). https://doi.org/10.1145/2339530.2339636

[33] Mohammed Javeed Zaki. 2001. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Mach. Learn.* 42, 1/2 (2001), 31–60.

[34] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (06 2014). https://doi.org/10.1145/2588555.2593671

A Comparative Evaluation of Anomaly Explanation Algorithms

Nikolaos Myrtakis*
University of Crete
Heraklion, Greece
myrtakis@csd.uoc.gr

Vassilis Christophides†
ENSEA, ETIS
Cergy, France
vassilis.christophides@ensea.fr

Eric Simon
SAP, France
Paris, France
eric.simon@sap.com

ABSTRACT

Detection of anomalies (i.e., *outliers*) in multi-dimensional data is a well-studied subject in machine learning. Unfortunately, unsupervised detectors provide no explanation about why a data point was considered as abnormal or which of its features (i.e. subspaces) exhibit at best its outlyingness. Such *outlier explanations* are crucial to diagnose the root cause of data anomalies and enable corrective actions to prevent or remedy their effect in downstream data processing. In this work, we present a comprehensive framework for comparing different unsupervised outlier explanation algorithms that are domain and detector-agnostic.

Using real and synthetic datasets, we assess the effectiveness and efficiency of two *point explanation* algorithms (Beam [28] and RefOut [18]) ranking subspaces that best explain the outlyingness of *individual* data points and two *explanation summarization* algorithms (LookOut [15] and HiCS [17]) ranking subspaces that best exhibit as *many outlier points* from inliers as possible. To the best of our knowledge, this is the first detailed evaluation of existing explanation algorithms aiming to uncover several missing insights from the literature such as: (a) Is it effective to combine any explanation algorithm with any off-the-shelf outlier detector? (b) How is the behavior of an outlier detection and explanation pipeline affected by the number or the correlation of features in a dataset? and (c) What is the quality of summaries in the presence of outliers explained by subspaces of different dimensionality?

1 INTRODUCTION

Detecting and diagnosing data anomalies are important tasks in data processing pipelines used to build industrial-strength Machine Learning (ML) systems [32]. Clearly, data points that significantly deviate from other points in a dataset may be systematic errors, i.e., *outliers*, or may manifest changes in the data generation process per se, i.e., *novelties*, that decrease the accuracy of the predictive models constructed downstream [29, 48]. In scientific and industrial monitoring applications, anomaly detection is often the ultimate goal of the data analysis as it enables the identification of unusual measurements (e.g., related to faults, bio-indices, etc.) and/or of suspicious activities (e.g. intrusions, fraud, etc.). Several unsupervised algorithms for anomaly detection have been proposed [2, 51] using different methods (e.g., proximity or isolation based) to distinguish outliers from inliers when labels of data points are impossible or difficult to obtain. Unfortunately, these algorithms do not explain why a data point was considered as abnormal, leaving analysts with no guidance about where to begin their investigation.

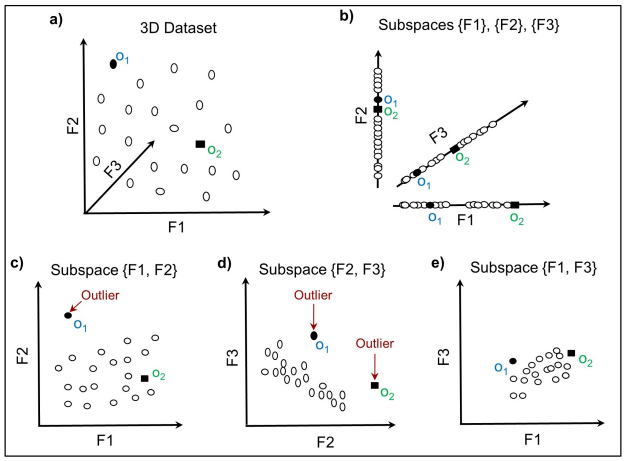


Figure 1: A 3*d* dataset with three 1*d* and 2*d* feature subspaces

In this paper, we focus on algorithms explaining the *outlyingness* aspects of multi-dimensional data points in the form of subspaces of data features that best explain why a given outlier deviates the most from the inliers. Such *explanations* are crucial to diagnose the root cause of data anomalies [3] and enable corrective actions to prevent or remedy their effect in downstream data processing (e.g. by repairing data errors or retraining the predictive models for concept drifts).

To illustrate, assume that we have a three dimensional dataset with features F_1, F_2 and F_3 and that we would like to explain the outlyingness of points $o1$ and $o2$ depicted by a black circle and a black square in Figure 1-a). In the full dimensional space of the dataset, $o1$ exhibits a small deviation from most of the other points in the dataset while $o2$ looks like an inlier although it exhibits a significant outlyingness when considering the subset of features $\{F_2, F_3\}$ (see Figure 1-d). We refer to the former case as *full space* outliers and to the latter as *subspace* outliers. In both cases, we are interested in explaining under which feature sets (aka subspaces) points exhibit a high outlyingness. None of the 1*d* subspaces $\{F_1\}$, $\{F_2\}$ and $\{F_3\}$ explain the outlyingness of the two points (see Figure 1-b). The same is true for the 2*d* subspace $\{F_1, F_3\}$ (see Figure 1-e). Subspace $\{F_1, F_2\}$ explains the outlyingness of $o1$ only (see Figure 1-c), while $\{F_2, F_3\}$ explains the outlyingness of both points (see Figure 1-d). We can observe that outlyingness of $o1$ is higher in $\{F_1, F_2\}$ than in $\{F_2, F_3\}$. Features contained into the explanation of an outlier are called *relevant*. For instance, F_1 and F_2 are relevant to the explanation of $o2$.

*Work was done while the author was working at SAP.

†This work received funding by the CY Initiative of Excellence (grant "Investissements d'Avenir" ANR-16-IDEX-0008) and developed during the author stay at the CY Advanced Studies, whose support is gratefully acknowledged.

We are primarily interested in *unsupervised* algorithms that are both *domain-agnostic* (i.e., suitable for datasets from various domains) and *detector-agnostic* (i.e., they can be employed to explain outliers produced by any off-the-self detector). Our choice of explanation algorithms is motivated by the fact that no detector is good in all possible settings w.r.t data characteristics (see the conclusions of several experimental studies [6, 8, 14]). Hence we are interested in decoupling the outlier explanation from detection, in contrast to several techniques proposed in Explainable Artificial Intelligence (XAI) such as output contribution of attribute values [24, 33] or partial dependence plots [11].

We evaluate two *point explanation algorithms*, *RefOut* [18] and *Beam* [28], that rank subspaces best explaining the outlyingness of *individual* data points, and two *explanation summarization algorithms*, *LookOut* [15] and *HICS* [17], that rank subspaces best explaining the outlyingness of as *many* outlier points as possible. These algorithms rely on outlyingness criteria of existing detectors such as Local Outlier Factor (LOF) [5], Angle Based Outlier Detection (ABOD) [21] or Isolation Forest (iForest) [23].

Although there exist several efforts for benchmarking outlier detectors in batch [6, 8, 12, 42] and stream [22, 43] processing settings, outlier explanation and summarization algorithms have not yet been thoroughly evaluated under realistic assumptions. To the best of our knowledge, this is the first comprehensive and detailed evaluation of existing algorithms aiming to uncover several insights missing from the existing literature. More precisely, our evaluation aims to answer the following questions: ***1.*** *Is it effective to combine any explanation algorithm with any off-the-shelf outlier detector?* ***2.*** *How is the behavior of an outlier detection and explanation pipeline affected by the number of features or their correlation in a dataset?* ***3.*** *What is the quality of summaries in the presence of outliers explained by subspaces of different dimensionality?*

The remaining of the paper is organized as follows. Section 2 introduces the outlier detectors and the point explanation and summarization algorithms we integrated in our experimental testbed. Section 3 details the pipelines of algorithms, the datasets as well as the evaluation metric used in our testbed. Section 4 presents the conducted experiments and the conclusions drawn regarding the missing insights. Section 5 surveys additional explanation algorithms for data in rest or in motion and justify why they have not been included in our study. Section 6 concludes our work and presents plans for future research.

2 OUTLIER DETECTION AND EXPLANATION ALGORITHMS

2.1 Unsupervised Outlier Detectors

Several methods have been proposed in the literature to measure the abnormality of a data point in a dataset. In the following, we survey three unsupervised methods that are widely used for detecting outliers in datasets with multiple numerical[1] features [6, 8, 12, 13, 42]. As the objective of outlier explanation is to retrieve subspaces where the outliers are clearly separable from inliers, we did not include any subspace-based outlier detector [20, 36] to assess the quality of a particular subspace examined by the explainer. The outlyingness criteria underlying each method have respective strengths and weaknesses w.r.t. the characteristics of the datasets (e.g., dimensionality) and outliers (e.g., highly clustered or not).

[1]Anomaly detection methods for categorical data [41] are outside the scope of this work.

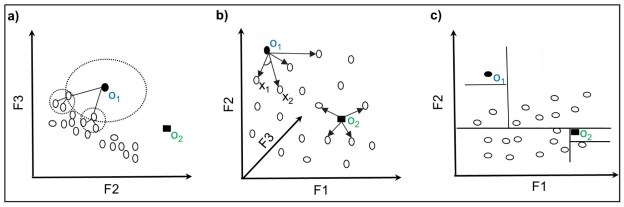


Figure 2: Examples of outliers in different subspaces detected by (a) LOF, (b) Fast ABOD and (c) iForest

Density-Based methods, such as Local Outlier Factor (LOF) [5] take into account the local density of points when searching for outliers. An example of outliers detected by LOF is illustrated in Figure 2-a). The point $o1$ is considered to be an outlier as it lies on a sparse area while its nearest neighbors lie on dense areas. The distance of a point p from o is computed using the following *reachability distance* (reach-dist):

$$\text{reach-dist}_k(p \leftarrow o) = max\{k\text{-dist}(o), d(p,o)\}$$

where k-dist(o) is the distance of o to its kth nearest neighbor and $d(p,o)$ is the direct distance (e.g., Euclidean) between the two points. LOF computes the local reachability density of a point p as the inverse of the average reachability distance of p from its k-nearest neighbors (kNN):

$$\text{lrd}_k(p) = 1/(\text{mean}_{o \in kNN(p)} \text{reach-dist}_k(p \leftarrow o))$$

Finally, the density of a point is compared to the average local reachability density of its neighbors to obtain a *score*:

$$\text{LOF}_k(p) = \text{mean}_{o \in kNN(p)} \frac{\text{lrd}_k(o)}{\text{lrd}_k(p)}$$

LOF's time complexity is $O(N^2)$, where N is the number of points in a dataset. Inliers obtain scores around 1 while outliers obtain scores significantly larger than 1. LOF distinguishes effectively outliers from inliers in regions of *varying density* where outliers lie on highly sparse areas far from dense clusters.

Angle-Based methods compute for each given point, the angles to other data points N. The Angle Based Outlier Detector (ABOD) [21] uses the variance of these angles as an outlyingness score. For example, as we can see in Figure 2-b), $o1$ is an outlier as its neighbors are located in similar directions (small angle variance), but o_2 is an inlier as it is surrounded by its neighbors in various directions (high angle variance). The ABOD score for a given point $o1$ and any pair of points $x1$, $x2$ is computed as:

$$ABOD(o_1) = \underset{x_1,x_2 \in N}{\text{Var}} \left(\frac{\langle \overrightarrow{x_1 o_1}, \overrightarrow{x_2 o_1} \rangle}{\|\overrightarrow{x_1 o_1}\|^2 \cdot \|\overrightarrow{x_2 o_1}\|^2} \right)$$

As ABOD's time complexity is $O(N^3)$, we are focusing on an efficient ABOD variant ($O(kN^2)$), called *Fast ABOD*, which computes the angles of a particular point only to its k-nearest neighbors. Small angle variance results to high ABOD score indicating high outlyingness. Intuitively, a point is more likely to be an outlier when it lies on the borders of the data distribution. ABOD avoids to compute the distance between points, hence it is a suitable detector for high dimensional datasets.

Isolation-Based methods estimate the probability of a point to be an outlier on the basis of the number of partitions needed to isolate it from the other points in a dataset. The less partitions needed to isolate, the more likely a data point is to be an outlier. For instance, in Figure 2-c) the point o_1 is an outlier as it needs less partitions to be isolated compared to the inlier o_2. Isolation Forest

(iForest) [23] exploits this property using a forest of random trees built on samples of the dataset by uniformly selecting features and their split values. The outlyingness score of a data point is then computed by averaging over all trees the path length from the root to the leaf node with the data point:

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}}$$

The score assigned to points is normalized within the range [0,1], with outliers getting a score close to 1. iForest has a small memory-footprint ($O(tn)$), where t is the number of trees and n is the subsample size. It achieves a sublinear time-complexity ($O(tnlogn)$) by exploiting subsampling and by eliminating the heavy cost of distance computation. Being agnostic to the distances (or densities) of points, iForest is able to detect outliers effectively even if they are lying on less dense areas than the majority of the points.

2.2 Point Explanation Algorithms

The objective of a point explanation algorithm is to discover the subspaces that best explain the outlyingness of a multi-dimensional point, i.e. the feature sets where this point deviates most in the dataset. Such subspaces are called *relevant* w.r.t. to the explanation of an outlier. Point explanation algorithms essentially rely on a *search strategy* for exploring feature subspaces in a dataset and an *outlyingness criterion*. The main challenge is that no interesting monotonic property holds for most outlyingness criteria [28], which prevents us to effectively prune the exponential space of feature sets (2^d) w.r.t. data dimensionality (d). Using the detectors presented previously, an outlier discovered in low-dimensional subspaces may become invisible, i.e., masked by inliers in high-dimensional subspaces and vice versa.

RefOut [18] is a sampling based algorithm which employs a stage-wise technique exploiting *random subspace projections* to find relevant subspaces of a *fixed dimensionality*. The main algorithmic steps of RefOut are illustrated in Figure 3. Initially, RefOut builds a random pool of size n with random subspace projections drawn from the full feature space of the dataset. In the example of Figure 3, we depict a pool of size 4 that contains $3d$ random subspaces (i.e, 50% of the $6d$ dataset). Using an off-the-self detector, the to-be-explained outlier $p1$ is scored in each subspace of the pool. To avoid dimensionality bias when scoring subspaces, the score of a point p in a subspace s, denoted as $score(p_s)$ is standardized using Z-score as follows:

$$score(p_s)' = \frac{score(p_s) - \overline{score_s}}{\sqrt{Var(score_s)}}$$

RefOut follows a stage-wise technique. In stage 1, RefOut assesses every single feature in the pool. In other words, in this stage it collects the best univariate subspaces. In our example, for the feature $F1$ RefOut partitions the pool into two populations of random subspaces w.r.t. whether they contain or not the feature $F1$. To assess the importance of a feature for explaining the outlyingness of the point $p1$, RefOut quantifies the discrepancy of score populations between the two partitions under the hypothesis that they have equal means. To test this hypothesis, the two-sample Welch's t-test [46] is employed as the two samples may have unequal variances and/or unequal sample sizes. The partitioning is repeated for every feature in the pool and the top-k ones with the highest discrepancy are kept; in our example we kept only $\{F1\}$ for simplicity. In stage 2, RefOut applies the same partitioning and scoring process for $2d$ subspaces by

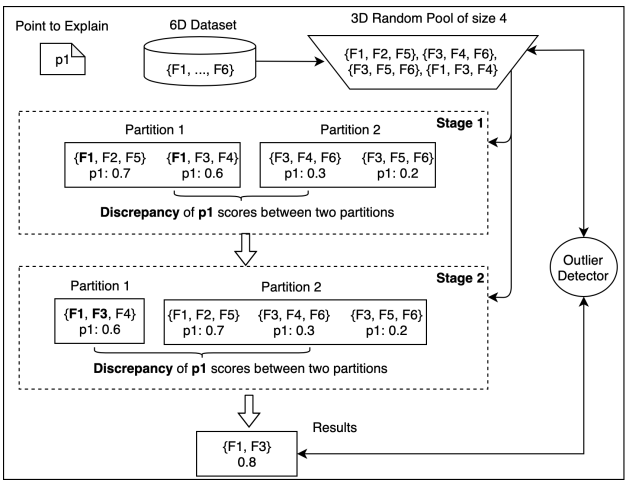


Figure 3: RefOut steps to find $2d$ subspaces from a $6d$ dataset to explain the point p1

taking the Cartesian product of the top-k subspaces from the previous stage with all the univariate subspaces drawn from the pool. In our example, since we are interested in $2d$ explanations the process stops at stage 2 and the best subspace ($\{F1, F3\}$) is returned as explanation of point $p1$. When multiple outliers have to be explained, RefOut searches for relevant subspaces for every point individually.

To sum up, the core idea of RefOut is to make subspace selection adaptive to the outlyingness score of each point and flexible w.r.t. different detectors. It relies on a pool of random subspace projections to assess the important features, that may contribute to the detection of relevant subspaces for a specific point. As feature importance is measured via the discrepancy of outlyingness score distributions, RefOut's effectiveness depends strongly on the ability of an off-the-self outlier detector to assign high scores to outliers. In particular, RefOut makes the assumption that outliers explained in low-dimensional subspaces exhibit a significant outlyingness also in their high-dimensional supersets.

Beam [28] is a *stage-wise* greedy algorithm that takes as input a particular point and returns the subspaces, up to a given dimensionality, that best explain its outlyingness. Although the maximum dimensionality of subspaces returned by Beam is predefined, the algorithm may output subspaces of varying dimensionality. Beam maintains two lists: (i) a *global list* of the best subspaces considered as relevant across stages, (ii) a *stage list* with the best subspaces in each stage. The main algorithmic steps of Beam are illustrated in Figure 4 via an example requesting to explain the outlyingness of a point $p1$ of a $6d$ dataset with up to $3d$ subspaces. Using an outlier detector, Beam scores exhaustively in stage 1 all the 15 $2d$ subspaces drawn from the 6 features space of the dataset for the point $p1$. Then, the top-k scored $2d$ subspaces will be inserted both into the *stage list* and *global list*. In stage 2, the best $2d$ subspaces kept in *stage list* will be combined with other features to form $3d$ subspaces as depicted in Figure 4. The top-k $3d$ subspaces are then kept in the *stage list*, while the *global list* is updated with the $3d$ subspaces with higher scores for $p1$ than the $2d$ subspaces previously computed. As we required $3d$ explanations in our example, the process will stop at stage 2. The *global list* is then returned as the result of the algorithm.

In a nutshell, Beam is a *stage-wise* greedy algorithm that exploits the top-k best relevant subspaces returned by early stages

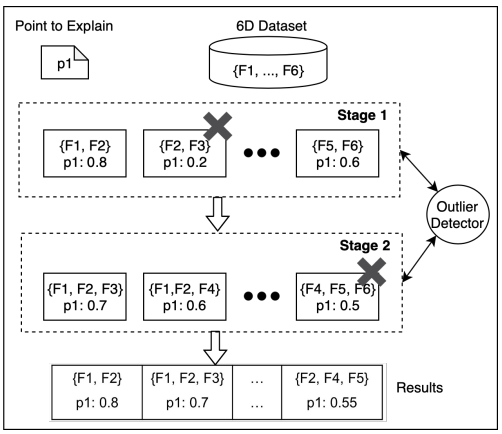


Figure 4: Beam steps to find subspaces up to 3 dimensions from a 6d dataset to explain the point p1

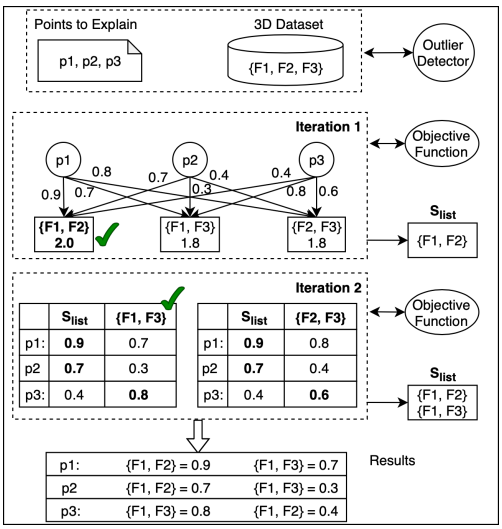


Figure 5: LookOut steps to find 2d subspaces from a 3d dataset with budget b = 2 (bold values indicate the highest scores per table row)

to search for relevant subspaces in latter stages. Hence, its effectiveness depends strongly on whether a given point obtains a high outlyingness score in *lower projections* of the relevant subspace(s) that should be finally returned. In order to make a fair comparison with RefOut, we report only the best subspaces from the *stage list* in the final stage i.e., subspaces of predefined maximum dimensionality. We call this variation $Beam_{FX}$.

2.3 Explanation Summarization Algorithms

The objective of an explanation summarization algorithm is to discover for a set of outlier points, the subspaces that distinguish as many outliers from inliers as possible. Explanation summarization algorithms also rely on a *search strategy* to explore feature subspaces in a dataset. The main difference is that the *outlyingness criterion* is applied *collectively* for all outliers rather than individually. The additional challenge stems from the fact that some outliers may be explained by subspaces of different dimensionality or in an extreme case all outliers could be explained by different subspaces. We should stress that explanation summarization is different from group identification and explanation. In the former case, we consider all the to-be-explained points as one group, while on the latter, the objective is to identify these anomalous groups and retrieve explaining subspaces that segregate each group from the normal instances [25].

LookOut [15] searches *exhaustively* subspaces of *fixed* dimensionality and returns those that exhibit a certain *utility*. LookOut was genuinely used to obtain 2d subspaces that can be easily visualized in order to explain a set of outliers. However, we used the algorithm to explore subspaces of high dimensionality as well. LookOut formalizes explanation summarization as maximization problem using an objective function equipped with the following properties: (i) *non-negative* , (ii) *non-decreasing* and iii) *sub-modular*. As submodular optimization is known to be an **NP-hard** problem, greedy approximation techniques are used (e.g., with a 63% approximation guarantee [27]). The main algorithmic steps of LookOut are depicted via an example in Figure 5. Given (i) a set of outlier points $P = \{p1, p2, p3\}$ and (ii) a number of top-k explanation summaries (i.e., the budget of the computation), LookOut constructs a subspace list S_{list} with the top-k subspaces that maximize the scores of the three points i.e., they provide a concise summary. Initially, LookOut employs an off-the-self outlier detector to score all outliers in the three possible 2d subspaces drawn from the 3d feature space of the dataset. LookOut's objective function for concise summarization is defined as follows:

$$f(S_{list}) = \sum_{p_i \in P} \max_{s_j \in S_{list}} score_{i,j}$$

where $score_{i,j}$ represents the outlier score that point p_i received in subspace s_j. Then, to assess utility of a subspace s to the S_{list}, LookOut examines its marginal gain computed as:

$$\Delta_f(s|S_{list}) = f(S_{list} \cup s) - f(S_{list})$$

In our example of Figure 5, S_{list} is initially empty and subspace $\{F1, F2\}$ is inserted during the first iteration as all three points obtain their best outlyingness score in this subspace. During the second iteration, LookOut examines which of the two remaining subspaces $\{F1, F3\}$ and $\{F2, F3\}$ provide the greatest marginal gain for S_{list}. In our example, $\{F1, F3\}$ has a higher marginal gain than $\{F2, F3\}$ as its maximizes $p3$'s score, while $p1$ and $p2$ scores are already maximized by $\{F1, F2\}$. The two subspaces are compared w.r.t. the maximum scores of every point currently in S_{list}. As the budget in our example is 2 i.e., the number of subspaces that will be included in explanation, the process stops and the S_{list} is returned as a summary of the subspaces explaining the points given as input.

In a nutshell, LookOut returns the top-k subspaces of fixed dimensionality that concisely explain multiple outliers. A subspace is considered a good summary candidate at a certain iteration step if it maximizes the overall score for at least one outlier. Hence, LookOut's effectiveness strongly depends on the ability of an off-the-self outlier detector to highly score outliers in their relevant subspaces.

High Contrast Subspaces (HiCS) [17] relies on a subspace search strategy that exploits combinations of correlated features called high contrast subspaces. The underlying intuition is that high contrast subspaces have many empty regions and few very dense regions, thus they are good candidates for separating outliers from inliers. Figures 6-a) to -c) illustrate three subspaces with correlated features ($\{F0, F1\}$, $\{F0, F1, F8\}$ and $\{F11, F12, F13\}$) while Figure 6-d) a subspace with non correlated features ($\{F11, F12\}$).

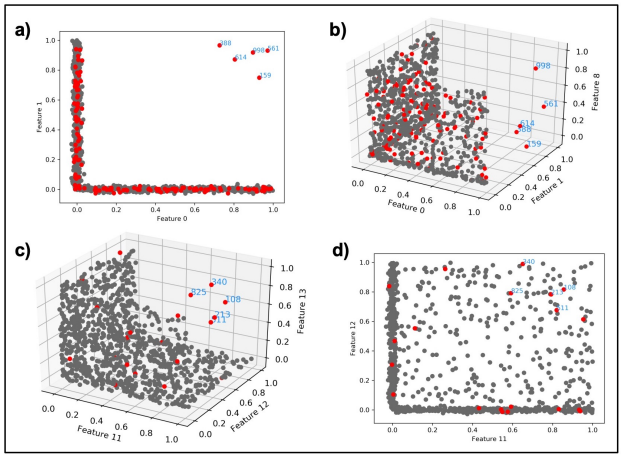


Figure 6: Data distribution in augmented/projected subspaces of HiCS Datasets

Subspace contrast in HiCS is measured using two-sample statistical tests[2] which are applied to the raw feature values under the null hypothesis that *both samples originate from the same underlying probability density function*. To enhance statistical precision, HiCS performs the statistical test for several Monte Carlo iterations and the average score is computed per subspace.

HiCS searches for high contrast subspaces via a stage-wise technique. In the first stage, it scores exhaustively all the 2d subspaces and selects the top-k based on their contrast. In next stage, the best 2d subspaces, are used to construct 3d subspaces scored again based on their contrast. The same procedure is repeated for several stages until reaching the full feature space d of a d-dimensional dataset; hence, the algorithm may retrieve subspaces of varying dimensionality. HiCS has been originally evaluated with LOF, but in principle any other off-the-self detector could be employed. In order to make a fair comparison with LookOut, we force HiCS to return subspaces of fixed dimensionality up to a predefined stage. We call this variation $HiCS_{FX}$.

To conclude, HiCS is a best effort algorithm that exploits subspaces with correlated features to discover summaries of varying dimensionality. Although the assumption that outliers are more likely to appear in correlated features seems effective for highly clustered anomalies, correlated subspaces may not always explain outliers, as depicted in Figure 1-e). The main novelty of HiCS lies in the decoupling of the subspace search strategy from the scores assigned by an off-the-self detector to a set of outliers.

3 BENCHMARKING ENVIRONMENT

The algorithms along with the datasets used in our testbed are available in our GitHub repository[3] to ensure repeatability of our experiments. Regarding outlier detectors, we used the implementation of LOF and iForest from Scikit-learn [30] and Fast ABOD from PyOD [50]. We have implemented LookOut, RefOut and Beam in java and modified HiCS implementation from ELKI [37]. Our primary concern in this work is the correctness of the implemented explanation algorithms. All experiments were performed in a Windows personal computer with a 4 core Intel i7 processor and 16GB of main memory.

[2]The Welch's t-test or the Kolmogorov-Smirnov test.
[3]https://git.io/JvuO6

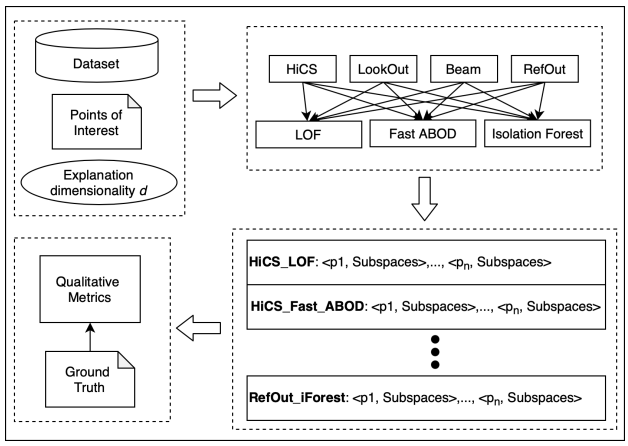


Figure 7: Pipelines of outlier detectors & explainers

3.1 Pipelines of Executed Algorithms

As illustrated in Figure 7, given (i) a dataset, (ii) a set of outliers (points of interest) and (iii) a target dimensionality to explain them, we execute all the possible pairs of explanation and detection algorithms. Each executed pipeline results to a list of *fixed*-dimensionality subspaces considered as relevant to each point of interest. The effectiveness of each pipeline is assessed using the relevant subspace(s) per point available in the ground truth of each dataset and the metric that we define in Section 3.3.

Regarding the choice of outlier detectors, we included in our testbed only LOF, Isolation Forest and Fast ABOD as representative of three widely used families of batch detection algorithms namely density, isolation and angle based outlier detection. As reported by several experimental studies [6, 8, 13] these algorithms frequently outperform distance or cluster-based algorithms in real and synthetic datasets while they do not require a thorough tuning of their hyper-parameters. Note also that experimentation with supervised detectors was outside the scope of our work due to the scarcity of labels regarding outlier/inlier data points.

To be able to retrieve the explaining subspaces for a number of outliers given as input, LookOut, Beam and RefOut heavily depend on the scores assigned by the detector in subspaces of different dimensionality explaining the given outliers. To shed some light regarding whether the explainers retrieve the relevant subspaces per outlier in practical settings, we employed unsupervised detectors that are not very sensitive to their hyper-parameter tuning. For LOF we use k = 15 and for Fast_ABOD k = 10. We run iForest for 10 repetitions to reduce the variance of outlyingness scores and the average score is computed for every point, using t = 100 trees and $sub-sample\ size$ = 256. These hyper-parameter values have been used in related experimental studies as [6], and allow us to detect the outliers in all datasets of our testbed. Hence, we can draw valuable conclusions for the subspace search techniques of explainers rather than the quality of the employed detector.

Regarding the hyper-parameters of the explainers, for HiCS we use $candidateCutOff$ = 400, a = 0.1, $Monte\ Carlo\ Iterations$ = 100 and Welch's t-test is performed. For LookOut we use $budget$ = 100. For Beam we use $beam-width$ = 100. For RefOut we use $poolsize$ = 100, $beam-width$ = 100, the random subspace dimensionality is set to 70% of dataset's dimensionality and Welch's t-test is performed. For HiCS, Beam and RefOut we return the top-100 subspaces as the final result.

Characteristics	Real Datasets (# 3)	Synthetic Datasets (# 5)
Outlier Type	Full Space	Subspace
Explanation Dimensionality	2-4 d	2-5 d
% Contamination with Outliers	10%	2, 3.4, 5.9, 10, 14.3 %
# Relevant Subspaces	60 (A), 151 (B), 249 (C)	4, 7, 12, 22, 31
# Relevant Subspaces per Outlier	3 (1 per dimensionality)	1 (91% outliers), 2 (9% outliers)
# Outliers per Relevant Subspace	1 (A), 1.13 (B), 1.45 (C)	5
% Relevant Feature Ratio	100%	35, 21, 12, 7, 5 %
Outlier Visibility w.r.t. Relevant Subspaces	Projections / Augmentations	Augmentations

Table 1: Characteristics of real and synthetic datasets

3.2 Real and Synthetic Datasets

In this section we describe the real and synthetic datasets used in our testbed. The main challenge in explaining outliers stems from the exponential search space of feature subspaces rather than the size of the dataset. The difference in the execution time of explainers depends more on the pruning strategy they employ to enumerate subspaces rather than on the time spent by detectors to score the explored subspaces. The selected datasets are suitable for assessing the quality of outlier explanation algorithms, as they provide the gold standard regarding the subspace(s) explaining each anomaly. To reduce confounding factors in the experimental evaluation of the algorithms, the selected datasets are mainly contaminated with *density-based* outliers. Outliers of this type can be detected by LOF but under certain conditions also by other detectors like ABOD and iForest (see Section 4). The main characteristics of our datasets are summarized in Table 1.

Breast, Breast Diagnostic and *Electricity Meter* are real-world datasets widely used to benchmark ML methods for anomaly detection [9]. To facilitate comparison with already published results, we used the version of these datasets[4] made available by the authors of RefOut algorithm [18]. Specifically, Breast (A) contains 198 points, 31 features and 20 outliers, Breast Diagnostic (B) contains 569 points, 30 features and 57 outliers and Electricity (C) contains 1205 samples, 23 features and 121 outliers. The ground truth provided per dataset contains the outliers detected by LOF resulting 10% contamination with outliers. Note that the experiments in [18] revealed that the reported outliers are *full space*. To obtain the best subspaces explaining them[5], we followed the method as described in [18] by performing an exhaustive search from 2 up to 4 dimensions for every dataset using LOF and keeping the top scored subspace per outlier at the corresponding dimension. We started from 2 dimensions as the initial step of HiCS and Beam perform an exhaustive search in 2d subspaces. We should stress that outliers are identifiable by LOF in both *lower dimensional projections* and *augmentations* (i.e., supersets) of the relevant subspaces. These datasets challenge summarization algorithms (HiCS and LookOut) as subspaces can best explain one outlier on average, e.g., for Electricity there are 1.43 outliers explained per relevant subspace (see Table 1).

HiCS synthetic datasets[6] were created by the authors of the HiCS [17] algorithm featuring *subspace outliers*. They initially splitted the datasets into 2d up to 5d subspaces, and generated high density clusters in each subspace. Then, they randomly picked 5 points and modified them to deviate from all clusters in each subspace. From these datasets we picked the dataset with the maximum dimensionality (100d) and splitted it into five sub-datasets from 14 up to 100 dimensions. The ratio of relevant features is depicted in Table 1 ordered from low (14d) to high (100d) number of features. Note that every dataset contains 1000 points. As illustrated in Figure 8 and Table 1, this split produced datasets of increasing (i) data dimensionality (i.e., number of features), (ii) number of relevant subspaces of different dimensionality and (iii) contamination with outliers. In HiCS datasets, the relevant subspaces and the outliers were given but there was no association between them. To identify the relevant subspace per outlier, we run LOF and keep the top-5 outliers with the highest scores per relevant subspace. The so obtained ground truth is aligned with the original contamination of the dataset with 5 points deviating in each relevant subspace that can be easily detected by LOF. An example of a 2d and a 3d relevant subspace is illustrated in Figures 6-a) and -c).

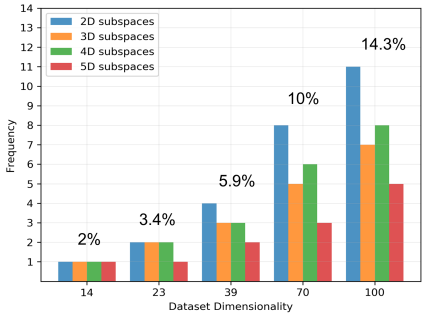


Figure 8: Dimensionality of subspaces relevant to outliers and contamination ratio of HiCS datasets

Note that the vast majority (~ 91%) of outliers in HiCS datasets is explained by one subspace and few outliers (~ 9%) by two different subspaces. These subspaces follow the properties: (i) they are disjoint in terms of features, (ii) each subspace can explain exactly five outlier points, (iii) they have highly correlated features, (iv) outliers are identifiable by the detectors in *augmented subspaces*, i.e., supersets of the relevant features (see example of Figures 6-a) and -b) and (v) outliers are mixed with inliers in *lower dimensional projections* of relevant subspaces (see example of Figures 6-c) and -d). Note that all outliers in HiCS datasets can be discovered by the three detectors used in our testbed.

3.3 Evaluation Metric

In this section we present the metric used to evaluate the effectiveness of the 12 pairs of outlier detection and explanation

[4]https://www.ipd.kit.edu/~muellere/RefOut/
[5]We discovered that the subspaces originally reported by the authors of RefOut were not optimal for most outliers.
[6]https://www.ipd.kit.edu/~muellere/HiCS/

algorithms (see Figure 7). Although outlier explanations target human analysts, we have not conducted user studies as our datasets are equipped with ground truth regarding which subspaces are relevant to the outliers they contain.

We denote the set of points of interest as P, the set of the relevant subspaces per point $p \in P$ as REL_p, and the returned subspaces from an explanation algorithm a to a point p as $EXP_a(p)$. The first metric we used is Mean Recall (see Eq. **??**) of an explainer a for a set of points P, assessing how many relevant subspaces were returned on average by a for every point $p \in P$. Note that a subspace in $EXP_a(p)$ is considered relevant only if it is a member of REL_p. I.e., a subspace in $EXP_a(p)$ is considered relevant for a point p only if it is identical with a subspace in REL_p. We consider only the Recall (see Eq. **??**) of points that are explained at a given dimensionality according to the ground truth. As described in Section 3.2, every point has very few relevant subspaces in our datasets. Thus high Mean Recall means that the explainer was able to exploit the relevant subspaces for the majority of the points explained at a given dimensionality. As each outlier in our datasets has very few relevant subspaces (specifically 1-3), we selected the MAP metric penalizing detectors that do not rank the relevant subspace(s) for an outlier within the top positions [40]. To compute MAP of an explainer a for a set of points P, we initially compute the precision (see Eq. 1) which is used to compute the Average Precision (see Eq. 2). $P@k(p)$ denotes the precision up to a k-th position of the returned subspaces in $EXP_a(p)$. The Boolean function $rel(k)$ indicates whether a subspace at the k-th position of $EXP_a(p)$ is relevant or not. Then, MAP is computed using the Average Precision of all points explained at a given dimensionality (see Eq. 3) according to the ground truth. A high MAP value indicates that for several points, the explainer was able to find and highly score their relevant subspaces using an outlier detector. Compared to other metrics such as accuracy, precision or recall, MAP better captures the scoring nature of outlier explanation algorithms: the discovered relevant subspaces should be ranked at the top positions of the list of candidates an algorithm considers. On the contrary, binary metrics like accuracy, precision and recall do not account for the ordering of the results. Note that [6, 8] use average precision to assess the quality of the outlier detector while [28] uses precision and recall to assess the explanation quality. To the best of our knowledge, it is the first work that relies on MAP to assess effectiveness of the subspace search strategies of different explainers.

$$\text{Precision}_a(p) = \frac{|REL_p \cap EXP_a(p)|}{|EXP_a(p)|} \tag{1}$$

$$\text{AveP}_a(p) = \frac{\sum_{k=1}^{|EXP_a(p)|} \text{P@k}(p) * rel(k)}{|REL_p|} \tag{2}$$

$$\text{MAP}_a(P) = \frac{1}{|P|} \sum_{p \in P} \text{AveP}(p) \tag{3}$$

4 EXPERIMENTS AND INSIGHTS

In this section we present our experiments for comparing point explanation and summarization algorithms. Our testbed includes the real datasets used in the evaluation of RefOut [18] as well as the synthetic datasets used in the evaluation of HiCS [17]. Both types of datasets were originally used to assess the effectiveness of detecting outliers hidden in subspaces rather than the suitability of the subspaces that led to the detection of those outliers. To the best of our knowledge, the only study investigating recall and precision of the subspaces of varying dimensionality retrieved by Beam was presented in [28] running on HiCS [17] datasets. In our study, we incorporate three more explainers, namely RefOut [18], HiCS [17] and LookOut [15], using also real world datasets. Moreover, in contrast to [28] we formulate different trade-offs by evaluating the pruning strategies under different explanation sizes as well as full and sub-space outliers.

4.1 Evaluation of Point Explanation Algorithms

The experiments of this section aim to answer two questions: (a) Is it effective to combine any explanation algorithm with any off-the-shelf outlier detector? (b) How is the behavior of outlier detection and explanation pipelines affected by the number of features in a dataset? To answer these questions, we run Beam and RefOut with LOF, Fast ABOD and iForest using the settings described in Section 3.1 for the synthetic and real-world datasets presented in Section 3.2. Figure 9 depicts for each dataset, the MAP (y-axis) of different outlier detection and explanation pipelines for explanations of increasing dimensionality (x-axis).

Figures 9-a) to -e) illustrate the MAP obtained in the five synthetic datasets of our testbed. Starting from the 14 dimensions in Figure 9-a), we observe that RefOut with LOF achieves optimal MAP as it retrieves and gives the highest score to the relevant subspaces for all the outliers, regardless of the explanation dimensionality. This is because (i) HiCS datasets contain highly clustered anomalies, thus LOF is the most suitable detector and (ii) the pool of RefOut contains low dimensional subspaces in which outliers can be more easily detected. Note that Beam with LOF has lower MAP for high explanation dimensionality since it does not retrieve all the relevant subspaces. Passing to 23 dimensions in Figure 9-b), the effectiveness of every pipeline drops especially for high dimensional explanations. RefOut with LOF seems to not be affected up to 3d explanations. An interesting behavior observed in this plot is that Beam is more effective with Fast ABOD and iForest than with LOF. This is due to the fact that the stage-wise strategy of Beam requires to collect lower dimensional projections of the relevant subspaces, so they could be formed in the final stage. Recall that in HiCS datasets, outliers are not separated from inliers in lower projections of the relevant subspaces (see Figure 6). According to complementary experiments not presented here due to space restrictions, in the early stages of Beam, the score distributions of outliers and inliers overlap less when Fast ABOD and iForest is used instead of LOF.

While the dimensionality of datasets increases, the same trends are observed in Figures 9-c) to -e). In general, Beam is able to retrieve all relevant 2d subspaces with the three detectors due to the exhaustive scoring of all feature pairs. However, its effectiveness starts dropping when the dimensionality of explanations increases. As the number of Beam stages increase, more subspaces need to be collected stage-wise with smaller differences in their score. RefOut proves to be more sensitive than Beam w.r.t. the number of features in the dataset D. As the dimensionality of random subspace projections in the pool is proportional to D's dimensionality, it becomes more difficult for RefOut to identify important features due to the less distinguishable score populations in subspaces. Observe that none of the algorithms seem to work for 4d explanations from 70 dimensions and higher and for 5d explanations from 23 dimensions and higher. Note that we run 10 times iForest (see Section 3.1) for every subspace considered

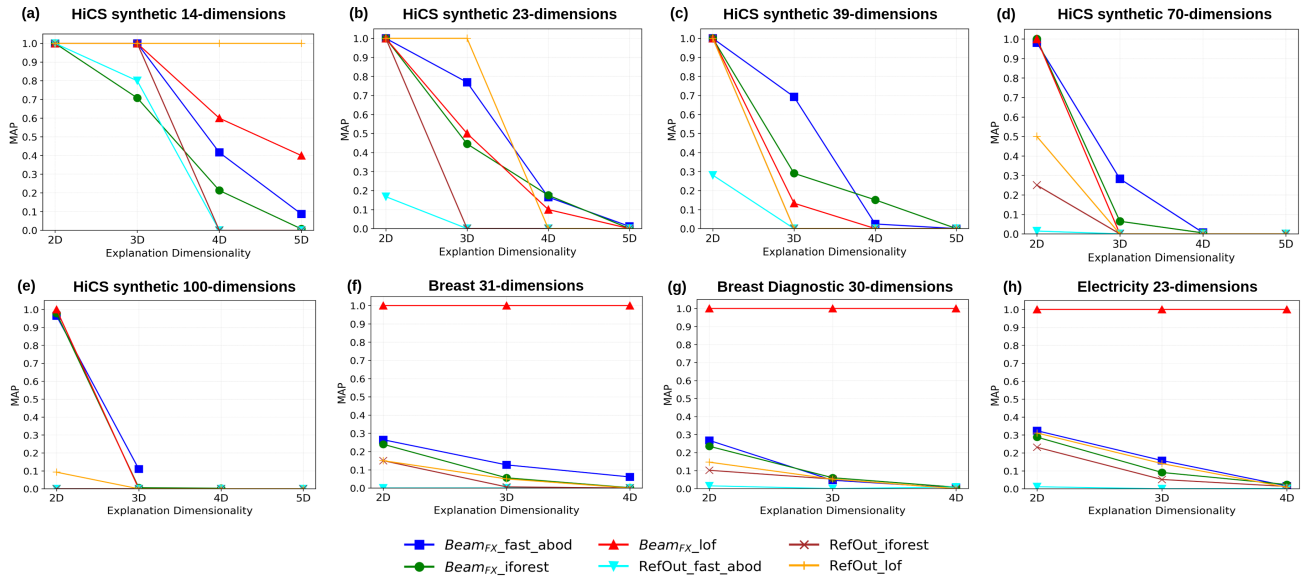


Figure 9: Mean Average Precision (MAP) of Beam and RefOut in HiCS synthetic datasets (a)-(e) and real-world datasets (f)-(h) for explanations of increasing dimensionality (best viewed in color)

by Beam up to 4d explanations for 70d and 100d datasets and Fast ABOD up to 4d explanations in 70d and up to 3d in 100d datasets. Specifically, to explain 100 outliers with 5d explanations in a 70d dataset, Beam needs to assess approximately 2.2M subspaces. In Section 4.3, we demonstrate that Beam requires an efficient detector such as LOF to assess a significant amount of subspaces.

Figures 9-f) to -h) illustrate the MAP obtained in the three real-world datasets of our testbed. Recall that in these datasets, the majority of the outliers are identifiable even in the full feature space. In general, Beam with LOF retrieves the optimal subspace for every outlier point (MAP = 1), despite of the explanation dimensionality. However, the effectiveness of Beam with Fast ABOD and iForest is significantly lower. On the contrary, RefOut seems to have very low MAP regardless of the employed detector. This is because RefOut cannot distinguish which features of full space outliers affect significantly the score populations generated by the corresponding detector.

Lessons Learned. Depending on the dataset characteristics, outlier detectors behave differently, affecting the effectiveness of explanation algorithms. A critical factor is whether outliers are masked by inliers in lower dimensional projections of the relevant subspaces (as in HiCS datasets). In this case, for datasets and explanations of low dimensionality, RefOut's random projection technique along with a detector suitable for the nature of outliers (e.g., LOF for clustered outliers) is preferred. For high dimensional datasets and low explanation dimensionality, Beam's stage-wise technique along with iForest or ABOD can effectively capture the small deviation of outliers in the subspaces considered by early stages. None of the algorithms seems to work for high explanation dimensionality (e.g., 4d and 5d) and high dataset dimensionality (e.g., 70d and 100d). When outliers are also visible in the full feature space (as in real-world datasets) the random projection technique exhibits poor MAP as it fails to find relevant features that significantly affect the score distributions. In this case, a stage-wise technique coupled with a suitable detector should be preferred regardless of the explanation dimensionality.

4.2 Evaluation of Summarization Algorithms

The experiments presented in this section aim to answer three questions: (a) Is it effective to combine any explanation summarization algorithm with any outlier detector?, (b) How is the behavior of outlier detection and explanation pipelines affected by the number of features or their correlation in a dataset?, and (c) What is the quality of summaries in the presence of outliers explained by subspaces of different dimensionality? To answer these questions, we run HiCS and LookOut with LOF, Fast ABOD and iForest using the settings described in Section 3.1 for the synthetic and real-world datasets presented in Section 3.2. Figure 10 depicts per dataset the MAP (y-axis) of different pairs of outlier detection and explanation algorithms for explanations of increasing dimensionality (x-axis). Despite the fact that HiCS does not use any detector to search candidate subspaces, it employs a detector to rank the retrieved subspaces. Thus, its effectiveness should be also evaluated for different detectors.

Figures 10-a) to -e) show the MAP of different algorithms for the five synthetic datasets of our testbed. Starting from 14 dimensions in Figure 10-a), HiCS and LookOut with LOF achieve optimal MAP regardless of the explanation dimensionality. As dataset's dimensionality and outlier ratio increase in Figures 10-b) to -e), HiCS with LOF and Fast ABOD are the most effective because (i) small groups of outliers are hidden within subspaces with correlated features and (ii) outliers are highly clustered at the borders of data distribution, allowing LOF and Fast ABOD to score their relevant subspaces at the top positions. The lowest MAP value of HiCS is observed in the 39 dimensional dataset where some 4d relevant subspaces do not contain highly correlated features. This drop clearly demonstrates the strong dependency of HiCS on the feature correlation heuristic.

As we can see in Figures 10-b and -e) LookOut's effectiveness significantly drops as the explanation dimensionality increases in higher dimensional datasets. One reason of this drop is related to the lower scores returned by the detectors in high dimensional subspaces. An additional reason stems from the existence of points exhibiting high outlyingness also in their augmented

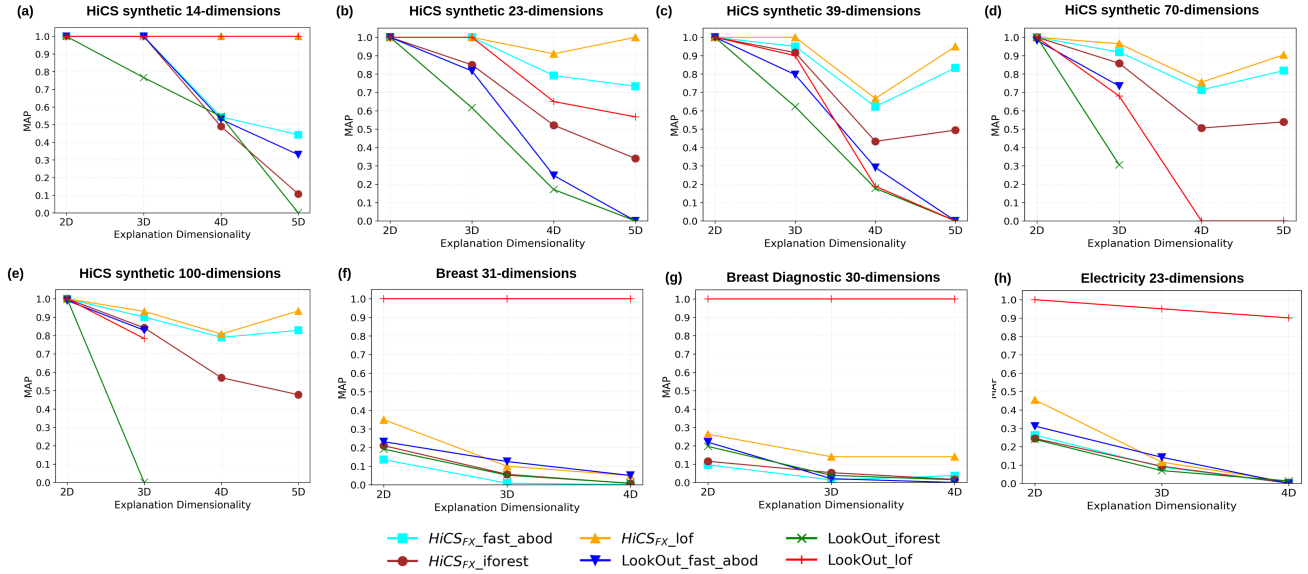


Figure 10: Mean Average Precision (MAP) of HiCS and LookOut in HiCS synthetic datasets (a)-(e) and real-world datasets (f)-(h) for explanations of increasing dimensionality (best viewed in color)

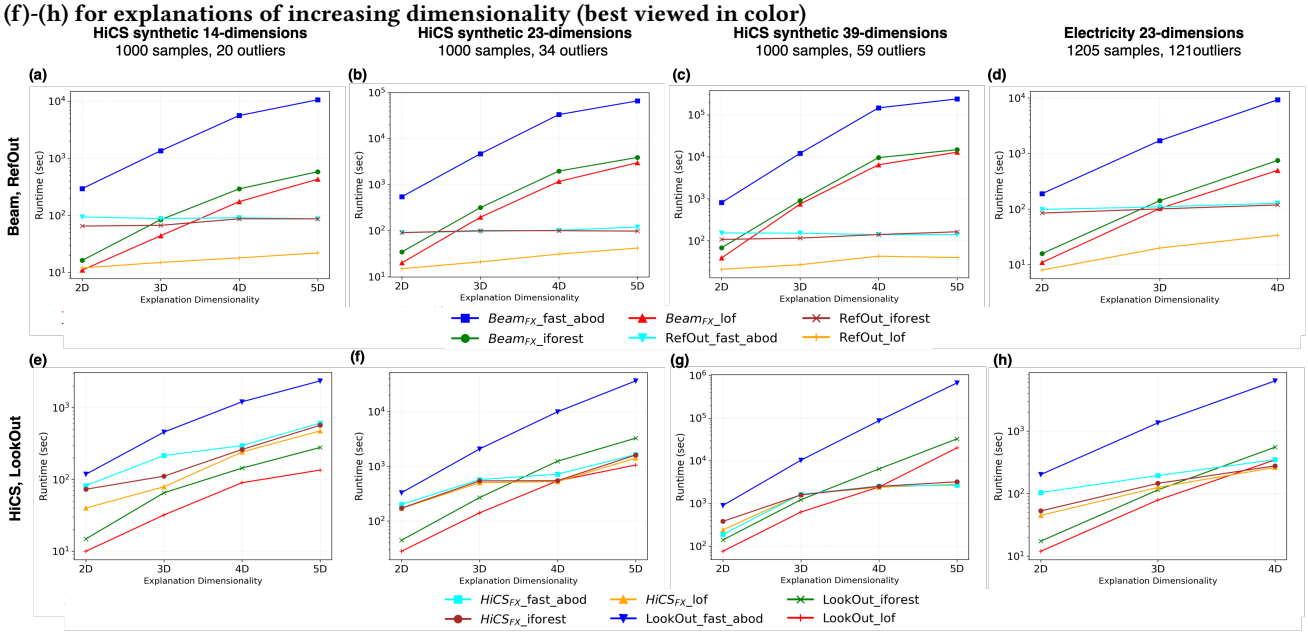


Figure 11: Runtime of detection and explanation pipelines (best viewed in color)

subspaces. According to complementary experiments not presented here due to space restrictions, detectors (especially LOF and iForest) assign higher scores to outliers in their augmented subspaces of dimensionality d than to outliers explained exclusively in d. As the outlier ratio increases along with dataset's dimensionality, more outliers get high scores in their augmented subspaces of a requested dimensionality. As a small fraction of outliers is explained by high dimensional subspaces, LookOut mainly retrieves augmented subspaces of outliers explained in lower dimensions that provide higher marginal gain. Observe that LookOut with Fast ABOD starts performing better than with LOF for high dataset dimensionality. Note that we run LookOut with LOF up to 4d explanations in 100 dimensions and Fast ABOD and iForest only up to 3d explanations for 70 and 100 dimensions. Specifically, to explain the outliers with 4d explanations in a 70d dataset, LookOut needs to assess 900K subspaces. In Section 4.3, we demonstrate that LOF is the most efficient detector when a significant amount of subspaces need to be assessed.

Figures 10-f) to -h) illustrate the MAP obtained in the 3 real-world datasets of our testbed. HiCS has poor MAP regardless of the explanation dimensionality or the detector used. This is because outliers are not contained in subspaces with highly correlated features. LookOut with LOF is the most effective as it is able to retrieve almost all relevant subspaces even when they maximally explain one outlier. On the contrary, LookOut with iForest and Fast ABOD exhibit poor performance as they are not able to highly score the relevant subspaces.

Lessons Learned. The fact that relevant subspaces may be formed by highly correlated features could be exploited to avoid a blind search of subspaces. When datasets exhibit strong feature correlation in relevant subspaces, HiCS exploits this heuristic and provides the best performance regardless of the dataset's

or explanation's dimensionality. It only depends on the ability of LOF or Fast ABOD to highly rank the retrieved subspaces. LookOut is as effective as HiCS in the synthetic datasets for low dataset dimensionality (e.g. 14d). When subspaces are formed by uncorrelated features, LookOut is a better alternative. However, LookOut is heavily impacted by the varying dimensionality of subspaces explaining different outliers. Indeed, the utility of subspaces in LookOut is defined exclusively in terms of their scores, without considering any semantic property of explanations such as the coverage of the points to be explained, or the overlap or the equivalence of subspaces in the explanation summaries.

4.3 Algorithms RunTime & Tradeoffs

In this section we report the execution time of the two point explanation and the two summarization algorithms we evaluated their effectiveness in Sections 4.1 and 4.2. In this respect, we are using the same synthetic (up to HiCS 39d) and real (Electricity 23d) datasets containing a similar amount of samples ($\sim$ 1000). We report execution time only for Electricity as it contains the highest number of samples exhibiting the same behavioral trends as the other two real datasets. Recall that as we are looking for explanations of fixed dimensionality (2-5d) the ratio of relevant features decreases as dataset's dimensionality increases.

Outlier Detection. Unlike HiCS, subspace search in explanation algorithms like Beam, RefOut and LookOut, heavily depends on the efficiency (and effectiveness) of used off-the-self detectors. According to the performance curves of detection and explanation pipelines depicted in Figure 11, LOF is the fastest followed by iForest and Fast ABOD across all datasets and explanation algorithms. This is due to low number of samples ($\sim$ 1000) despite the fact that iForest has the lowest time complexity. A similar result has been reported in [8] for the same hyper-parameter settings as those used in our testbed (see Section 3.1). Note that for iForest we report the average time out of 10 repetitions per subspace. Specifically, to score a single subspace LOF needed 0.05, iForest 0.2 and Fast ABOD 2 seconds approximately.

Point Explanation. The runtime of pipelines involving Beam, RefOut are illustrated in Figures 11-a) to -d). Critical factors affecting Beam's efficiency are: (i) the requested explanation dimensionality (more stages to be built), (ii) the dataset's dimensionality (more subspaces to be assessed per stage), (iii) the efficiency of the employed detector and (iv) the number of outliers to explain (the process is repeated per outlier). However, due to its random sampling technique, RefOut's runtime is relatively stable regardless of the explanation or dataset's dimensionality. Note that up to 39d datasets and 2d explanations, RefOut and Beam with LOF need almost the same time to assess a similar amount of subspaces. RefOut with LOF outperforms Beam with LOF from 1 (in real datasets) up to 3 orders (in synthetic datasets) of magnitude for 39d datasets and 5d explanations.

Explanation Summarization. The runtime of pipelines involving LookOut and HiCS are illustrated in Figures 11-e) to -h). The critical factors affecting LookOut's efficiency are: (i) dataset's and explanation dimensionality (exhaustive subspace search) and (ii) the efficiency of the employed detector. On the other hand, by decoupling subspace search from outlier scoring, the critical factor of HiCS efficiency is only the explanation dimensionality (more subspaces to be assessed per stage). Thus, HiCS exhibits similar running times when executed with LOF, iForest and Fast ABOD (used only to rank the discovered subspaces). Surprisingly, LookOut with LOF[7] outperforms all HiCS pipelines up to 4d explanations (by 1 order of magnitude in 2d). For the size of datasets used in our experiments, HiCS statistical tests to assess feature correlation prove to be more costly than LOF distance calculation of points to assess their outlyingness. Performance gains of LookOut with LOF drop as we increase the number of features along with explanation dimensionality, leading HiCS to outperform LookOut in the 39d dataset for 5d explanations.

Table 2 demonstrates the point explanation and summarization algorithms along with their corresponding detector that exhibit the best tradeoff between effectiveness (according to Figures 9 and 10) and efficiency (according to Figure 11) from 2d up to 5d explanations across decreasing relevant feature ratios. For every cell we take the top pair of algorithms according to their efficiency and effectiveness in pareto order. We prioritize generic algorithms like LookOut over algorithms like HiCS that work under specific conditions. For instance, LookOut with LOF is slightly less effective than HiCS with LOF in Figure 10-c), while they have the same execution time in Figure 11-g). In cells 2d and 3d with a 12% ratio, we consider that LookOut achieves a better tradeoff since it is more generic than HiCS. When point explanation or summarization algorithms exhibit zero effectiveness in all executed pipelines for a particular dataset and explanation dimensionality, no top pair is reported. For instance, for 5d and 21% or 12% ratios only one pair for detection and summarization algorithms is reported (HiCS with LOF) as no point explanation algorithm succeeds to return relevant 5d explanations. The main conclusions drawn from Table 2 are:

1. *State-wise subspace search* employed by Beam achieves the best tradeoff for full space outliers. Both its effectiveness and efficiency significantly decrease for subspace outliers as the ratio of relevant features decreases. However, it is the only option for high explanation dimensionality (3d - 4d) and low relevant feature ratio ($<$ 12%).

2. *Random subspace projection* employed by RefOut provides a good tradeoff for subspace outliers with a medium ratio of relevant features (35% and 21%). Its effectiveness drops to zero as the explanation dimensionality becomes greater than 3d (for 21% ratio).

3. *Exhaustive subspace search* employed by LookOut exhibits top effectiveness and efficiency for full space outliers regardless of the explanation dimensionality, as well as, for subspace outliers up to 3d. Its effectiveness significantly drops for subspace outliers explained by subspaces greater than 3d (for 21% ratio).

4. *Correlation heuristic* exploited by HiCS achieves the best tradeoff for 4d-5d explanations especially when the relevant feature ratio is low. This heuristic however, strongly depends on the data distribution as highly clustered outliers may are not always be visible in correlated features.

5 RELATED WORK

In this section we survey additional explanation algorithms for data in rest (databases) or in motion (streams) and justify why they have not included in our benchmark.

Explaining Black-Box Models. Several methods have been proposed to explain why a supervised model predicted a particular label for a particular example [10, 19, 24, 26, 33]. LIME [33] constructs a linear interpretable model that is locally faithful to the

[7] LookOut has been experimentally evaluated by its authors [15] only with iForest and 2d explanations.

Explanation Dimensionality	Relevant Features Ratio			
	100%	**35%**	**21%**	**12%**
2*d*	Beam LOF LookOut LOF	RefOut LOF LookOut LOF	RefOut LOF LookOut LOF	RefOut LOF LookOut LOF
3*d*	Beam LOF LookOut LOF	RefOut LOF LookOut LOF	RefOut LOF LookOut LOF	Beam Fast Abod LookOut LOF
4*d*	Beam LOF LookOut LOF	RefOut LOF LookOut LOF	Beam iForest HiCS LOF	Beam iForest HiCS LOF
5*d*	Beam LOF LookOut LOF	RefOut LOF LookOut LOF	HiCS LOF	HiCS LOF

Table 2: Tradeoffs of outlier detection and explanation algorithms

predictor. [10, 19] explain the model by perturbing the features to quantify their influence on predictions. Other works aim to produce explanations in the form of feature relevance scores by comparing the difference between a classifier's prediction score and the score when a feature is assumed to be unobserved [34], or by considering the local gradient of the classifier's prediction score with respect to the features for a particular example [4]. [38, 39] considered how to score features in a way that takes into account the joint influence of feature subsets on the classification score. This body of work requires as input a supervised model rather than an unsupervised anomaly detector. However, in real application settings it is difficult or even impossible to label data as anomalous or normal examples [12].

Explaining Outliers in Query Answers. Scorpion [47] was the first system for explaining outliers in the result of group-by queries. Given a set of outliers spotted by analysts on the results of queries, the system searches for a logical formulae that describes a set of tuples that contribute most to the excessively high or low aggregate value of a specific group. It is hard to extend this work for explaining outliers recognized by off-the-self detectors. Furthermore, empirical explanations for data points that violate specific data quality constraints (i.e., inconsistencies w.r.t. domain-specific rules) have been studied in [7]. A glitch explanation is a collection of values of features that have statistically significant propensity signatures. In our work, we are interested in a quantitative form of data anomalies frequently encountered in transaction or measurement-based datasets, i.e., outliers in numerical features for which quality constraints are difficult or impossible to obtain. Finally, an interactive explanation discovery system has been proposed [35]. It relies on a set of explanation templates given by analysts that need to be precomputed per dataset. Neither of the previous methods satisfy our requirements for explaining data anomalies in a way that is both domain and detector agnostic without making strong assumptions regarding how the input datasets have been processed.

Explaining Outliers in Temporal Data. MacroBase [1] enables efficient, accurate, and modular analyses that highlight and aggregate important and unusual behavior in fast data. It introduces an operator for explaining outliers in a data stream based on the categorical features rather than the numerical features used to actually detect outliers. In contrast to the notion of relevant subspaces, the explanation of continuous outliers consists of conjunctions of categorical features whose values cover most of the outliers detected by a density-based method called MAD. ExplainIT [16] is a recent system for unsupervised root-cause analysis of time series that shares similar motivations with MacroBase. It empowers a declarative interface (SQL based) for specifying a large number of cause hypothesis that need to be tested and ranked to assist analysts with a reduced number of causal dependencies that have to exploit regarding an observed phenomenon. The use of causal models for explaining data outlyingness is an interesting idea that we plan to study in the future by leveraging our previous work on scalable algorithms for causal feature discovery [45]. Finally, EXstream [49] is a system providing high-quality explanations for anomalous behaviors of streaming data that analysts annotate using CEP-based monitoring results. Explanations take the form of logical formulae in CNF involving relational predicates (i.e., $=$, $<$, $\leq$) over feature values computed for time series. Authors formalize the problem of optimally explaining anomalies in CEP as an information reward maximization problem. In this respect, an entropy-based distance function of time series is used to measure the contribution in the reward of each feature. As the reward function is sub-modular, greedy approximation techniques could be used as in the case of LookOut [15]. Computing explanations based on single-feature rewards bears similarity with the univariate feature selection problem while computing subspace based outlier explanations is closer to the more complex problem of multivariate feature selection [45].

6 CONCLUSIONS AND FUTURE WORK

In this experimental study, we addressed missing insights regarding the performance of existing outlier explanation and summarization algorithms under realistic settings. We underlined the main challenge that stems from the lack of inherent pruning properties to effectively search the exponential space. Existing subspace search strategies exploit the distributional characteristics either: (i) of data such as features' correlation in subspaces (HiCS [17]) or (ii) of scores given by an outlier detector in subspaces (LookOut [15], Beam [28] and RefOut [18]). The former strategy is effective when highly clustered outliers over correlated features are contained in datasets regardless of their dimensionality, while the latter is effective in low explanation dimensionality where the outlier detectors can discriminate accurately the outliers from the inliers. It remains open to assess whether the low dimensional subspaces retrieved by an explainer are projections of a high dimensional subspace fully explaining a specific point.

We should additionally note that the detection of outliers in LOF, ABOD and iForest, is actually decoupled from the search of subspaces likely to contain them. HiCS, RefOut and Beam instead are *explaining outlier detectors* that rely on per-subspace measures to quantify the explanation quality of subspaces. We are planning to extend our testbed with recent works [44] taking into account the relationship between subspaces using a dimension-based measure of their explanation quality. Moreover, in case of recurring anomaly patterns, it is also interesting to benchmark group-based explanation summarization techniques [25].

Another interesting aspect would be to investigate outlier explanation in stream processing settings such as LODA [31].

We should finally stress that existing outlier explanation and summarization algorithms actually provide *descriptive explanations*. In essence, subspace explanations are verbose descriptions of the decision boundary discovered by unsupervised detectors to distinguish inliers from outliers. This is the reason why explanation tasks should be re-executed for every new bunch of data made available in ML pipelines even if they stem from the same generative process. As summarization algorithms can only exploit the subspaces which are assessed to be relevant to a given set of points, they may result in summaries of very poor quality when individual outliers are explained by disjoint feature subsets. In this respect, we are planning to build a surrogate model to predict the scores (or labels) of points produced by an unsupervised outlier detector and approximate its decision boundary using minimal predictive signatures. Such *predictive explanations* overcome the high computation cost of subspace search per point and provide formal guarantees regarding minimality in the explanation dimensionality.

REFERENCES

[1] F. Abuzaid, P. Bailis, J. Ding, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Sahaana. Macrobase: Prioritizing attention in fast data. *ACM Trans. Database Syst.*, 43(4):15:1–15:45, Dec. 2018.

[2] C. C. Aggarwal. *An Introduction to Outlier Analysis*, pages 1–40. Springer New York, 2013.

[3] H. Aguinis, R. K. Gottfredson, and H. Joo. Best-practice recommendations for defining, identifying, and handling outliers. *Organizational Research Methods*, 16(2):270–301, jan 2013.

[4] D. Baehrens, T. Schroeter, S. Harmeling, M. Kawanabe, K. Hansen, and K. Müller. How to explain individual classification decisions. *J. Mach. Learn. Res.*, 11:1803–1831, 2010.

[5] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: Identifying density-based local outliers. In *SIGMOD Conference*, 2000.

[6] G. O. Campos, A. Zimek, J. Sander, R. J. Campello, B. Micenkova, E. Schubert, I. Assent, and M. E. Houle. On the evaluation of unsupervised outlier detection: Measures, datasets, and an empirical study. *Data Min. Knowl. Discov.*, 30(4):891–927, July 2016.

[7] T. Dasu, J. M. Loh, and D. Srivastava. Empirical glitch explanations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 572–581. ACM, 2014.

[8] R. Domingues, M. Filippone, P. Michiardi, and J. Zouaoui. A comparative evaluation of outlier detection algorithms: Experiments and analyses. *Pattern Recognition*, 74:406 – 421, 2018.

[9] D. Dua and C. Graff. UCI machine learning repository, 2017.

[10] R. C. Fong and A. Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. In *ICCV*, pages 3449–3457, 2017.

[11] J. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2001.

[12] M. Goldstein and S. Uchida. A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. *PLoS One*, 11(4), 4 2016.

[13] X. Gu, L. Akoglu, and A. Rinaldo. Statistical analysis of nearest neighbor methods for anomaly detection. In *NeurIPS*, pages 10921–10931, 2019.

[14] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi. A survey of methods for explaining black box models. *ACM Comput. Surv.*, 51(5), Aug. 2018.

[15] N. Gupta, D. Eswaran, N. Shah, L. Akoglu, and C. Faloutsos. Beyond outlier detection: Lookout for pictorial explanation. In *ECML/PKDD*, 2018.

[16] V. Jeyakumar, O. Madani, A. Parandeh, A. Kulshreshtha, W. Zeng, and N. Yadav. Explainit! – a declarative root-cause analysis engine for time series data. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 333–348. ACM, 2019.

[17] F. Keller, E. Müller, and K. Böhm. Hics: High contrast subspaces for density-based outlier ranking. *ICDE*, pages 1037–1048, 2012.

[18] F. Keller, E. Müller, A. Wixler, and K. Böhm. Flexible and adaptive subspace search for outlier analysis. In *CIKM*, 2013.

[19] P. W. Koh and P. Liang. Understanding black-box predictions via influence functions. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1885–1894. PMLR, 2017.

[20] H.-P. Kriegel, P. Kröger, E. Schubert, and A. Zimek. Outlier detection in axis-parallel subspaces of high dimensional data. In *PAKDD*, 2009.

[21] H.-P. Kriegel, M. Schubert, and A. Zimek. Angle-based outlier detection in high-dimensional data. In *KDD*, pages 444–452. ACM, 2008.

[22] A. Lavin and S. Ahmad. Evaluating real-time anomaly detection algorithms – the numenta anomaly benchmark. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 38–44, Dec 2015.

[23] F. T. Liu, K. M. Ting, and Z.-H. Zhou. Isolation forest. *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, 2008.

[24] S. M. Lundberg and S. Lee. A unified approach to interpreting model predictions. In *NeurIPS*, pages 4765–4774, 2017.

[25] M. Macha and L. Akoglu. Explaining anomalies in groups with characterizing subspace rules. *Data Min. Knowl. Discov.*, 32(5):1444–1480, Sept. 2018.

[26] G. Montavon, W. Samek, and K. Müller. Methods for interpreting and understanding deep neural networks. *Digit. Signal Process.*, 73:1–15, 2018.

[27] G. L. Nemhauser and L. A. Wolsey. Best algorithms for approximating the maximum of a submodular set function. *Math. Oper. Res.*, 3:177–188, 1978.

[28] X. V. Nguyen, J. Chan, S. Romano, J. Bailey, C. Leckie, K. Ramamohanarao, and J. Pei. Discovering outlying aspects in large datasets. *Data Mining and Knowledge Discovery*, 30:1520–1555, 2016.

[29] A. Nurunnabi and G. West. Outlier detection in logistic regression: A quest for reliable knowledge from predictive modeling and classification. In *2012 IEEE 12th International Conference on Data Mining Workshops*, pages 643–652, Dec 2012.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[31] T. Pevný. Loda: Lightweight on-line detector of anomalies. *Machine Learning*, 102:275–304, 2015.

[32] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. Data lifecycle challenges in production machine learning: A survey. *SIGMOD Rec.*, 47(2):17–28, 2018.

[33] M. T. Ribeiro, S. Singh, and C. Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *KDD*, pages 1135–1144, 2016.

[34] M. Robnik-Sikonja and I. Kononenko. Explaining classifications for individual instances. *TKDE*, 20:589–600, 2008.

[35] S. Roy, L. Orr, and D. Suciu. Explaining query answers with explanation-ready databases. *Proc. VLDB Endow.*, 9(4):348–359, Dec. 2015.

[36] S. Sathe and C. C. Aggarwal. Subspace outlier detection in linear time with randomized hashing. *ICDM*, pages 459–468, 2016.

[37] E. Schubert and A. Zimek. Elki: A large open-source library for data analysis - elki release 0.7.5 "heidelberg". *ArXiv*, abs/1902.03616, 2019.

[38] E. Strumbelj and I. Kononenko. An efficient explanation of individual classifications using game theory. *J. Mach. Learn. Res.*, 11:1–18, 2010.

[39] E. Strumbelj and I. Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowl. Inf. Syst.*, 41(3):647–665, 2014.

[40] W. Su, Y. Yuan, and M. Zhu. A relationship between the average precision and the area under the roc curve. In *ICTIR '15*, 2015.

[41] A. Taha and A. S. Hadi. Anomaly detection methods for categorical data: A review. *ACM Comput. Surv.*, 52(2), May 2019.

[42] K. M. Ting, T. Washio, J. R. Wells, and S. Aryal. Defying the gravity of learning curve: a characteristic of nearest neighbour anomaly detectors. *Machine Learning*, 106:55–91, 2016.

[43] L. Tran, L. Fan, and C. Shahabi. Distance-based outlier detection in data streams. *Proc. VLDB Endow.*, 9(12):1089–1100, Aug. 2016.

[44] H. Trittenbach and K. Böhm. Dimension-based subspace search for outlier detection. *International Journal of Data Science and Analytics*, 7(2):87–101, Mar 2019.

[45] I. Tsamardinos, G. Borboudakis, P. Katsogridakis, P. Pratikakis, and V. Christophides. A greedy feature selection algorithm for big data of high dimensionality. *Machine Learning*, 108(2):149–202, 2019.

[46] B. L. Welch. The significance of the difference between two means when the population variances are unequal. *Biometrika*, 29(3/4):350–362, 1938.

[47] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *Proc. VLDB Endow.*, 6(8):553–564, June 2013.

[48] H. Xiong, G. Pandey, M. Steinbach, and V. Kumar. Enhancing data analysis with noise removal. *TKDE*, 18(3):304–319, Mar. 2006.

[49] H. Zhang, Y. Diao, and A. Meliou. Exstream: Explaining anomalies in event stream monitoring. In *EDBT*, pages 156–167, Mar. 2017.

[50] Y. Zhao, Z. Nasrullah, and Z. Li. Pyod: A python toolbox for scalable outlier detection. *J. Mach. Learn. Res.*, 20:96:1–96:7, 2019.

[51] A. Zimek and P. Filzmoser. There and back again: Outlier detection between statistical reasoning and data mining algorithms. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, 8(6), 2018.

Scaling Density-Based Clustering to Large Collections of Sets

Daniel Kocher
University of Salzburg
Salzburg, Austria
dkocher@cs.sbg.ac.at

Nikolaus Augsten
University of Salzburg
Salzburg, Austria
nikolaus.augsten@sbg.ac.at

Willi Mann
Celonis SE
Munich, Germany
w.mann@celonis.com

ABSTRACT

We study techniques for clustering large collections of sets into DBSCAN clusters. Sets are often used as a representation of complex objects to assess their similarity. The similarity of two objects is then computed based on the overlap of their set representations, for example, using Hamming distance. Clustering large collections of sets is challenging. A baseline that executes the standard DBSCAN algorithm suffers from poor performance due to the unfavorable neighborhood-by-neighborhood order in which the sets are retrieved. The DBSCAN order requires the use of a symmetric index, which is less effective than its asymmetric counterpart. Precomputing and materializing the neighborhoods to gain control over the retrieval order often turns out to be infeasible due to excessive memory requirements.

We propose a new, density-based clustering algorithm that processes data points in any user-defined order and does not need to materialize neighborhoods. Instead, so-called backlinks are introduced that are sufficient to achieve a correct clustering. Backlinks require only linear space while there can be a quadratic number of neighbors. To the best of our knowledge, this is the first DBSCAN-compliant algorithm that can leverage asymmetric indexes in linear space. Our empirical evaluation suggests that our algorithm combines the best of two worlds: it achieves the runtime performance of materialization-based approaches while retaining the memory efficiency of non-materializing techniques.

1 INTRODUCTION

We consider the problem of partitioning large collections of sets into DBSCAN [15] clusters. Our work is motivated by a process mining use case at Celonis SE that models processes as sets. A *process* is a sequence of timestamped activities. Large companies store hundreds of millions of activities in millions of processes. In order to analyze the processes, they should be clustered into groups of similar activity sequences that can be further explored [5, 22, 39]. To this end, a process is represented by the set of all its neighboring activity pairs, e.g., the process with the activity sequence (S, O, P, H, R, F, E) (Start, Order, Pay, sHip, Return good, reFund, End) is represented by the set $\{(S, O), (O, P), (P, H), (H, R), (R, F), (F, E)\}$. The similarity of two processes is then assessed by the Hamming distance[1] of their set representations.

Sets are used in many other applications [29] to represent objects for the purpose of clustering, e.g., sales may be represented by sets of product categories, photos by sets of tags and title words, user interactions on a website by sets of visited links, users of a social network by their group memberships, or users of a music streaming platform by sets of tracks they listen to.

[1]Hamming distance $H(r, s) = |r \cup s| - |r \cap s|$ for two sets r and s.

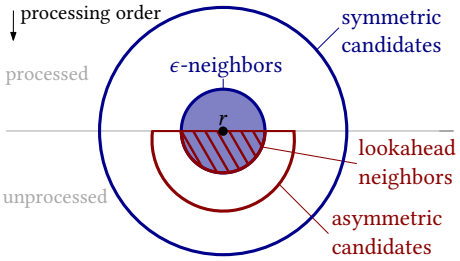


Figure 1: Symmetric candidates with ϵ-neighbors (blue); asymmetric candidates with lookahead neighbors (red).

The popular DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm [15] identifies clusters of arbitrary shape without requiring the number of clusters as input. Intuitively, DBSCAN finds dense regions that are separated by regions of lower density. The density of a region (given a distance function between pairs of data points) is defined by two parameters: the number of neighbors, *minPts*, and the radius, ϵ, of the neighborhood. A data point is called *core point* (i.e., it is at the core of a dense region) if it has at least minPts neighbors (including itself) within radius ϵ; a non-core point in the ϵ-neighborhood of a core point is a *border point* (i.e., it is at the border of a dense region); all other points are *noise* [37].

The runtime of the DBSCAN algorithm heavily depends on the efficiency of the neighborhood computation. In our experiments, the neighborhood computation accounts for up to 99% of the overall runtime for some datasets. Therefore, in order to efficiently cluster large collections of sets, effective indexing techniques for sets are required.

Similarity indexes for sets have been proposed in the context of ϵ-neighborhood joins, which are executed in an index nested loop fashion. A prominent representative is the *prefix index* [1, 8], which is linear in size and has been shown to be highly effective [29]. The *symmetric* prefix index returns the complete ϵ-neighborhood for a given query point r. The *asymmetric* prefix index assumes a processing order on the sets in R and retrieves only the *lookahead neighbors*: the ϵ-neighbors that follow r in processing order. A typical processing order for sets is based on the set sizes (ties broken arbitrarily). The *asymmetric* prefix index further leverages the length information to avoid many of the candidates that the symmetric index must inspect (among the unprocessed sets). Figure 1 illustrates the ϵ-neighborhood, the lookahead neighbors, and the candidate regions of symmetric and asymmetric index, respectively. The region above the gray line represents the sets that have been processed before r, the region below the gray line are unprocessed sets. The circles and semicircles show subset relationships.

Clearly, the asymmetric prefix index is preferable in terms of effectiveness over its symmetric counterpart. Unfortunately, there is an inherent mismatch between the asymmetric index and the DBSCAN algorithm. DBSCAN suffers from the following issues when executed with the asymmetric index: (1) *Core status*

problem: The lookahead neighbors of r are not sufficient to update the core status of r. (2) *Border vs. Noise problem:* To distinguish border points from noise, a border point must be visible from a core point, which is not guaranteed by the asymmetric lookahead neighborhood. (3) *Disconnected clusters:* To guarantee connected clusters, DBSCAN imposes a (partial) processing order on the neighborhood computations: all core points of the current cluster must be expanded (i.e., their neighborhood must be computed) before any point belonging to a different cluster is processed.

A well-known clustering approach [3] is based on a self-join that precomputes and materializes all neighborhoods. The precomputed neighborhoods are then used while executing DBSCAN. This approach can leverage the asymmetric index and is efficient in runtime. Unfortunately, this join-based technique requires quadratic memory in the worst case and suffers from a large memory footprint in practice. For example, for our social media dataset (LIVEJ) that stores the interests of 3.1M users, this approach requires almost 100GB of memory.

Summarizing, applications that must cluster large collections of sets have two options, which we call Sym-Clust and Join-Clust. (1) Sym-Clust: Retrieve the full ϵ-neighborhoods in the processing order imposed by DBSCAN using the symmetric index. (2) Join-Clust: Compute and materialize neighborhoods in a join using the effective asymmetric index. None of the options is satisfying: Sym-Clust runs almost up to an order of magnitude slower than Join-Clust, while Join-Clust is infeasible for many datasets and parameter settings due to its excessive memory usage.

We propose a new clustering algorithm, *Spread*, that computes correct DBSCAN clusters using the asymmetric prefix index. Spread runs in linear space and does not need to materialize the (quadratic-size) neighborhoods. Spread avoids symmetric neighbor computations, therefore reducing the number of neighbors retrieved by Sym-Clust. So-called *backlinks* are introduced to achieve a correct clustering. Backlinks are dynamically added and removed as required and occupy only a small fraction of the memory that is used by materialized neighborhoods. Spread maintains a graph of subclusters in a disjoint-set data structure and guarantees that connected components in the resulting graph represent correct DBSCAN clusters.

In general, Spread can process data points in any *user-defined order* given an index that retrieves the lookahead neighbors, i.e., all data points that follow the query point in the user-defined processing order. In our usage scenario – set clustering – the processing order is defined by the set sizes (ties broken arbitrarily) and the asymmetric prefix index retrieves lookahead neighbors.

Summarizing, our contributions are the following:

- We propose *Spread*, a novel algorithm for partitioning large collections of sets into DBSCAN clusters. To the best of our knowledge, this is the first linear space DBSCAN-compliant algorithm that leverages the *asymmetric* prefix index for sets.
- We introduce the new concept of *backlinks* that keep sufficient information to build correct clusters independently of the processing order that the user imposes on Spread. We prove invariants for backlinks and the correctness of our approach.
- Our extensive empirical evaluation on 13 real-world datasets suggests that Spread is as fast as Join-Clust (that materializes all neighborhoods) while being competitive in memory usage with Sym-Clust (that computes all neighborhoods on the fly).

The remainder of this paper is organized as follows. In Section 2, we cover the background on ϵ-neighborhood and set similarity, indexing techniques for sets, and density-based clustering, and we define the *density-based set clustering problem*. Section 3 presents the two baseline approaches for density-based set clustering, Join-Clust and Sym-Clust. In Section 4, we present Spread, our time- and space-efficient solution for density-based set clustering. We evaluate our solution against the baseline algorithms and discuss the results in Section 5. Related work is summarized in Section 6. Finally, Section 7 concludes this paper.

2 BACKGROUND & PROBLEM DEFINITION

We revisit set similarity indexes and density-based clustering, and define our problem. To simplify the presentation, we focus on prefix indexes for the Hamming distance. Our results, however, extend to other distance and similarity measures (e.g., Jaccard or Cosine) and the respective indexes [12, 29, 40]. The required adaptations of the index that have been studied in the context of set similarity joins [29, 49] (e.g., prefix length, size lower bound, equivalent overlap) also apply to our scenario.

2.1 Set Similarity and ϵ-Neighborhood

R is a collection of n unique sets, each set $r \in R$ consists of unique tokens $t_1, \ldots, t_m$, $|r| = m$. The *processing order*, $\succ$, is a total order defined over R. The similarity between two sets r and s is assessed by the Hamming distance, $H(r, s) = |r \cup s| - |r \cap s|$, which counts the number of tokens that exist only in one of the sets, e.g., $H(r_1, r_2) = 4$ and $H(r_2, r_3) = 3$ for the sets in Figure 2.

The *ϵ-neighborhood* of set r includes r and all sets within distance ϵ from r, $N_\epsilon(r) = \{s \in R \mid H(r, s) \leq \epsilon\}$. A *region query* on r computes $N_\epsilon(r)$. A set r splits its ϵ-neighborhood into two disjoint parts based on the processing order: the *lookahead neighbors* that follow r in processing order, $N_\epsilon^{\succ}(r) = \{s \in N_\epsilon(r) \mid s \succ r\}$ and the *preceding neighbors*, $N_\epsilon^{\prec}(r) = \{s \in N_\epsilon(r) \mid s \prec r\}$.

2.2 Indexing Techniques for Sets

Prefix Filter and Inverted Index. A naive approach computes a region query $N_\epsilon(r)$ by verifying the predicate $H(r, s) \leq \epsilon$ for all sets $s \in R$. An effective indexing technique, which was originally developed for set similarity joins [2, 29], is based on the so-called prefix filter. The prefix, π_r, of a set r consists of the first π tokens of r according to some total token order (which must be the same for all sets). The prefix length depends on the distance function and is $\pi = \epsilon + 1$ for the Hamming distance. Figure 2 shows the prefix of three sets for distance threshold $\epsilon = 3$ and a numerical token order. The prefix filter works best if the tokens in the prefix are infrequent, thus the tokens are typically ordered by ascending global token frequency.

A set $s \in R$ can be in the ϵ-neighborhood $N_\epsilon(r)$ only if the prefixes of r and s share at least one token, i.e., $H(r, s) \leq \epsilon \Rightarrow \pi_r \cap \pi_s \neq \emptyset$ (assuming $|r| + |s| > \epsilon$; otherwise r and s are always similar). Therefore, if two sets do not share a token in the prefix, the pair can be safely pruned. If two sets r and s share a prefix token, (r, s) is a *candidate pair* and must undergo *verification*, i.e., the predicate $H(r, s) \leq \epsilon$ must be evaluated. Candidates that fail verification are *false positives*. Mann et al. [29] discuss efficient prefix-based verification.

Symmetric Prefix Index. An inverted index on the prefix tokens is used to retrieve candidate pairs efficiently. The inverted index maps prefix tokens to sets that contain that token in the prefix. A lookup of set r retrieves all lists of the prefix tokens of r. The

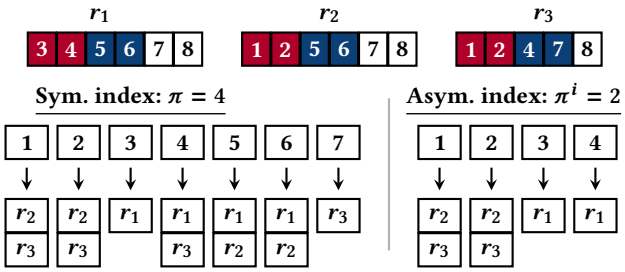


Figure 2: Symmetric and asymmetric prefix index, $\epsilon = 3$.

union of these lists (except r itself) are the candidates of r. The index is *symmetric* and returns the ϵ-neighborhood of r.

For example, the candidates for r_2 returned by the symmetric index, $\pi = \epsilon + 1 = 4$, in Figure 2 are $\{r_1, r_3\}$ (resulting from the union of $[r_2, r_3]$ for token 1, $[r_2, r_3]$ for token 2, $[r_1, r_2]$ for token 5, and $[r_1, r_2]$ for token 6). Candidate r_1 is a false positive since $H(r_1, r_2) > \epsilon$; r_3 is a true positive due to $H(r_2, r_3) \leq \epsilon$.

Asymmetric Prefix Index. We construct an *asymmetric* prefix index that returns only the lookahead neighbors, $N_\epsilon^{\succ}(r)$. To this end, we define a length-based processing order on R (longest to shortest): r precedes s if $|r| > |s|$, i.e., $|r| > |s| \Rightarrow s \succ r$; ties ($|r| = |s|$) are broken by the lexicographic order of the sorted sets.

Since we are interested only in sets $s \in N_\epsilon(r)$ that are no larger than r, $|s| \leq |r|$, we need to index only a subset of the prefix tokens: the tokens in the so-called *indexing prefix* [49]. The so-called *probing prefix* of the lookup set, r, remains of length $\pi = \epsilon + 1$. For the Hamming distance, the indexing prefix is of length $\pi^i = \left\lfloor \frac{\epsilon}{2} + 1 \right\rfloor$. For $\epsilon > 0$, the probing prefix is always longer than the indexing prefix, e.g., $\pi = 4$ and $\pi^i = 2$ for $\epsilon = 3$. A shorter prefix results in fewer candidates and renders the asymmetric index more effective than its symmetric counterpart.

In the case of r_2, the asymmetric index, $\pi^i = 2$, in Figure 2 returns only a true positive candidate, r_3. The false positive candidate, r_1, which is returned by the symmetric index, is avoided.

2.3 Density-Based Clustering

We formally define DBSCAN clusters and the related concepts. A set r represents a point to be clustered. The *density* of r is the number of ϵ-neighbors $|N_\epsilon(r)|$ (cf. Section 2.1).

Core, Border, Noise. A set r is a *core point* iff the ϵ-neighborhood of r contains at least minPts sets: r is core $\Leftrightarrow |N_\epsilon(r)| \geq$ minPts. A set s is a *border point* iff it is in the ϵ-neighborhood of a core point r and s is not core: s is border $\Leftrightarrow s \in N_\epsilon(r) \wedge |N_\epsilon(r)| \geq$ minPts $\wedge$ $|N_\epsilon(s)| <$ minPts. All remaining sets in R are *noise*. We denote the set of core and border points with $\mathcal{C}$ and $\mathcal{B}$, respectively. The set of noise points is $\mathcal{N} = R \setminus (\mathcal{C} \cup \mathcal{B})$.

Density-Reachability. Let $r, s \in R$ and r is core: s is *directly density-reachable* from r iff s is in the ϵ-neighborhood of r: $r \blacktriangleright s \Leftrightarrow s \in N_\epsilon(r)$. If there is a sequence of sets $r_1, r_2, \ldots, r_k$ with $r_1 = r$ and $r_k = s$, $r_i \blacktriangleright r_{i+1}$ for $1 \leq i < k$, s is *density-reachable* from r, denoted $r \blacktriangleright \ldots \blacktriangleright s$. Two sets r, s are *density-connected* if there is a set x s.t. both r and s are density-reachable from x.

A *density-based cluster* is a subset $C_i \subseteq R$ that satisfies two criteria [38]:

(1) *Maximality* $\mathbb{M}$*:* For any two sets $r, s \in R$, $r \in C_i$. If s is density-reachable from r, then $s \in C_i$. Formally,

$$\forall r, s \in R : r \in C_i \wedge r \blacktriangleright \ldots \blacktriangleright s \implies s \in C_i$$

Table 1: Notation overview.

Notation	Description
R	a collection of sets
r, s, x	sets of R
$\|r\|$	cardinality of set r
$r \prec s, r \succ s$	r precedes/succeeds s (in R)
$H(r, s)$	the Hamming distance of two sets r, s
π, π^i	probing/indexing prefix
ϵ	distance threshold
minPts	minimum density s.t. a set r is core
$N_\epsilon(r)$	full ϵ-neighborhood of r
$N_\epsilon^{\prec}(r), N_\epsilon^{\succ}(r)$	preceding/lookahead neighbors of r
$r \blacktriangleright s$	s is directly density-reachable from r
$r \blacktriangleright \ldots \blacktriangleright s$	s is density-reachable from r
$\mathcal{C}, \mathcal{B}, \mathcal{N}$	the set of core, border, and noise sets
C_i	a density-based cluster with id i

(2) *Connectivity* $\mathbb{C}$*:* For any two sets r, s in C_i, there is a set x that density-connects r and s. Formally,

$$\forall r, s \in R : r, s \in C_i \implies \exists x \in C_i : r \blacktriangleleft \ldots \blacktriangleleft x \blacktriangleright \ldots \blacktriangleright s$$

DBSCAN Clustering. A border point may be part of multiple density-based clusters such that the clusters overlap. We define the *DBSCAN clustering* that partitions the data into non-overlapping clusters. The standard DBSCAN algorithm [15] produces a DBSCAN clustering.

Definition 2.1. Let $R^* = R \setminus \mathcal{N}$ and $C_1, C_2, \ldots, C_k$ be density-based clusters such that $\bigcup_{i=1}^{k} C_i = R^*$. A DBSCAN clustering is a partitioning $\Gamma = \{C'_1, C'_2, \ldots, C'_k\}$, $C'_i \subseteq C_i$, such that $\bigcup_{i=1}^{k} C'_i = R^*$, $C'_i \cap C'_j = \emptyset$ for $i \neq j$.

A *subclustering* of a cluster C_i, $\psi_i = \{c_1, c_2, \ldots, c_l\}$, is a partitioning of C_i into $1 \leq l \leq |C_i|$ non-empty, disjoint subclusters, $c_j \subseteq C_i$, such that $\bigcup_{j=1}^{l} c_j = C_i$, $c_j \cap c_k = \emptyset$ for $j \neq k$.

A *subcluster graph* of R^* is an undirected graph in which nodes are subclusters and an edge between two nodes can only exist if the respective nodes are in the same DBSCAN cluster.

2.4 The DBSCAN Algorithm

The standard DBSCAN algorithm [15] forms clusters by repeatedly picking a seed point from the set of unvisited data points (initially all points are unvisited). If the seed is a core point, it forms a new cluster with all points that are density-reachable from the seed and are not yet assigned to a cluster. The set of density-reachable points is computed by recursively adding the ϵ-neighbors of all core points to the current cluster. The algorithm terminates when all points have been visited. Points that cannot be assigned to a cluster are noise.

2.5 Problem Statement

Definition 2.2 (Density-Based Set Clustering). Given a collection of sets R, a distance threshold ϵ, and the neighborhood density minPts, the goal is to find a DBSCAN clustering $\Gamma = \{C_1, C_2, \ldots, C_k\}$ of R.

For sets, asymmetric indexes with a lookahead neighbor function $N_\epsilon^{\succ}(r)$ promise the best performance (cf. Section 2.2). Given an ordering $\succ$ on R, we strive for a time- and space-efficient algorithm that solves the density-based set clustering problem with an asymmetric index.

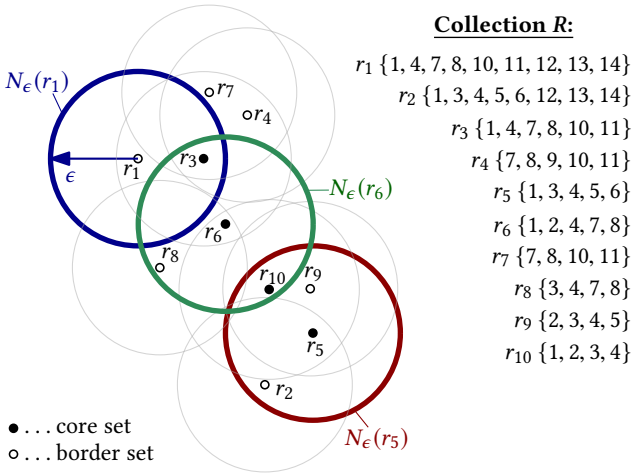


Figure 3: Running example, $\epsilon = 3$, minPts = 4.

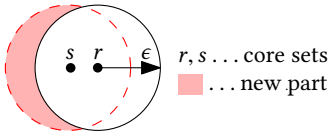


Figure 4: Redundant neighborhood queries.

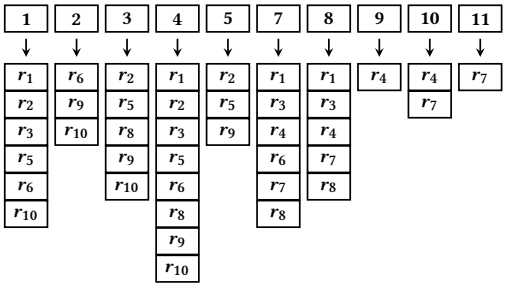


Figure 5: Symmetric prefix index on r_1-r_{10}, $\epsilon = 3$, $\pi = 4$.

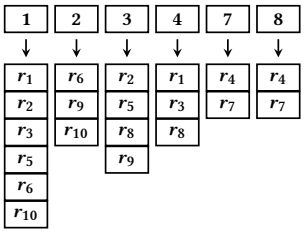


Figure 6: Asymmetric prefix index on r_1-r_{10}, $\epsilon = 3$, $\pi^i = 2$.

Running Example. Figure 3 shows an example collection R of ten sets, r_1–r_{10}, and their neighborhoods for Hamming distance $\epsilon = 3$. Sets r_3, r_5, r_6, and r_{10}, are core sets; all sets r_1–r_{10} form a single cluster.

3 BASELINE APPROACHES

This section presents two baseline solutions for the density-based set clustering problem. (1) *Sym-Clust* is memory-efficient and follows the standard DBSCAN approach with the symmetric prefix index to answer neighborhood queries on the fly. (2) *Join-Clust* is speed-optimized and materializes all ϵ-neighborhoods using a state-of-the-art set similarity join algorithm [2] (which leverages the asymmetric prefix index) before the standard DBSCAN algorithm is executed.

Both baselines leverage state-of-the-art set indexes. We are not aware of other previous solutions that can outperform Sym-Clust or Join-Clust for the density-based set clustering problem. Note that using the standard DBSCAN [15] (rather than some advanced techniques presented in later works, cf. Section 6) is not a limiting factor: Most of the overall execution time is spent computing the neighborhoods, and prefix-based indexes are highly efficient in combination with efficient verification [29].

3.1 Sym-Clust: DBSCAN with Inverted Index

When the standard DBSCAN algorithm (cf. Section 2.4) picks a seed point that is core, it forms a cluster with all points that are density-reachable from the seed. The density-reachable points are computed by pushing all core neighbors of the seed onto a stack. Then, each point on the stack is processed in the same manner (i.e., all its core neighbors are pushed onto the stack) until the stack is empty. All neighbors of core points retrieved in this process belong to the cluster.

The neighborhood queries will overlap to some extent. Assume r is processed before s, $s \in N_\epsilon(r)$, then $|N_\epsilon(s) \cap N_\epsilon(r)| \geq 2$ (at least r and s are in both neighborhoods). Since r assigns all its neighbors to the current cluster, only the non-overlapping neighbors of s, $N_\epsilon(s) \setminus N_\epsilon(r)$, will further increase the cluster.

Figure 4 illustrates this observation for the neighborhoods of two example points r (black circle) and s (red circle): only the new, non-overlapping area of $N_\epsilon(s)$ (shaded in red) is relevant for expanding the cluster.

The standard DBSCAN algorithm requires the use of a symmetric index since it assumes to see all neighbors of a point s when s is processed. The asymmetric index is not compatible with the standard DBSCAN algorithm: We cannot impose an order on the points such that all non-overlapping neighbors of s follow s in the processing order. Further, the size of the neighborhood of s, $|N_\epsilon(s)|$, is required to decide its core status.

Figure 5 shows the symmetric prefix index for our running example, $\epsilon = 3$, $\pi = \epsilon + 1 = 4$. We probe $r_8 = \{3, 4, 7, 8\}$. The prefix of r_8 consists of all tokens in r_8 (due to $|r_8| = \pi$). The union of the respective index lists yields the candidates $\{r_2, r_5, r_9, r_{10}, r_1, r_3, r_6, r_4, r_7\}$. Note that the candidates include both sets that are smaller and sets that are larger than r_8.

The so-called *length filter* [2], an optimization of the symmetric prefix index that also applies to its asymmetric counterpart, prunes candidates r_2 and r_1. Due to their length difference to r_8, they cannot be in the ϵ-neighborhood of r_8. By ordering the lists in processing order (i.e., longer sets precede shorter sets, as illustrated in Figure 5), the length filter can prune the head (sets that are too long) and the tail (sets that are too short) of a list without inspecting all elements in head and tail, respectively.

All candidates that are not pruned by the length filter must undergo verification. Only r_6 passes verification, therefore $N_\epsilon(r_8) = \{r_8, r_6\}$, and r_8 is classified as non-core (minPts = 4).

Complexity Analysis. We probe each set $r \in R$ against the index once. With cost C for an index lookup and $n = |R|$, the runtime is $O(n \cdot C)$; $C = O(n)$ since r may have $O(n)$ neighbors, thus the overall runtime of Sym-Clust is $O(n^2)$. The symmetric index is of linear size leading to space complexity $O(n)$.

Algorithm 1: Materialize-Neighborhoods(R, ϵ)

Input: Collection of sets R, distance threshold ϵ
Result: Materialized neighborhoods of R w.r.t. ϵ

$I \leftarrow$ Create-Index (R, ϵ);
$pairs \leftarrow \emptyset$ // Result set of similar pairs
foreach $r \in R$ *in processing order* **do**
  $M \leftarrow$ Probe (r, I, ϵ) // candidates with prefix overlaps
  foreach $(s, po) \in M$ **do** // po . . . prefix overlap
    if Verify-Pair (r, s, ϵ, po) **then**
      $pairs \leftarrow pairs \cup \{(r, s)\}$
$neighborhoods \leftarrow$ new associative array of size $|R|$;
foreach $(r, s) \in pairs$ **do**
  $neighborhoods[r] \leftarrow neighborhoods[r] \cup \{s\}$;
  $neighborhoods[s] \leftarrow neighborhoods[s] \cup \{r\}$;
return *neighborhoods*

Algorithm 2: Create-Index (R, ϵ)

Input: Collection of sets R, distance threshold ϵ
Result: Inverted index of R w.r.t. ϵ

$I \leftarrow \emptyset$ // inv. index of set prefixes, $I_{r[p]}$ . . . list of token $r[p]$
foreach $r \in R$ **do**
  $\pi \leftarrow \left\lfloor \frac{\epsilon+2}{2} \right\rfloor$ // indexing prefix length of r
  for $p \leftarrow 1$ **to** π **do** $I_{r[p]} \leftarrow I_{r[p]} \cup \{r\}$;
return I

Algorithm 3: Probe (r, I, ϵ)

Input: Probing set r, inv. index I, distance threshold ϵ
Result: Set of candidates for r w.r.t. ϵ

// M maps a candidate s to its prefix overlap with r
$M \leftarrow$ new associative array // candidates
$\pi \leftarrow \epsilon + 1$ // probing prefix length of r
$lb_r \leftarrow |r| - \epsilon$ // size lower bound wrt. r
for $p \leftarrow 1$ **to** π **do**
  foreach $s \in I_{r[p]}$ *in proc. order* **do** // list of token $r[p]$
    if $|s| < lb_r$ **then break** ;
    else // add candidate
      if $s \notin M$ **then** $M[s] \leftarrow 0$; // init.
      $M[s] \leftarrow M[s] + 1$ // incr. overlap of (r, s)
return M

Algorithm 4: Verify-Pair (r, s, ϵ, po)

Input: Probing set r, candidate set s, distance threshold ϵ, prefix overlap po
Result: True iff r and s are similar w.r.t. ϵ, false otherwise

// cf. Mann et al. [29] for prefixes, equiv. overlaps, and Verify proc.
$\pi_r, \pi_s \leftarrow$ probing resp. indexing prefix length of r resp. s;
$w_r, w_s \leftarrow \pi_r$- resp. π_s-th token in r resp. s;
$t \leftarrow$ equivalent overlap for r, s, and ϵ;
if $w_r < w_s$ **then**
  return Verify $(r, s, t, po, \pi_r + 1, po + 1)$
return Verify $(r, s, t, po, po + 1, \pi_s + 1)$

3.2 Join-Clust: Materialized Neighborhoods

Join-Clust executes a set similarity self-join on R and materializes the ϵ-neighborhoods in main memory. The self-join traverses all sets of $r \in R$ in processing order and computes their lookahead neighbors, $N_\epsilon^\succ(r)$. The lookahead neighbors of r are appended to the list of r's neighbors, and r is appended to the neighborhood lists of all $s \in N_\epsilon^\succ(r)$. After processing all sets, the neighborhood list of each set $r \in R$ is complete and stores $N_\epsilon(r)$.

Next, standard DBSCAN (cf. Section 2.4) is executed to form clusters using the materialized neighborhoods. Algorithms 1–4 implement the similarity join with neighborhood materialization, index creation, probing, and efficient verification [29].

Mann et al. [29] found that the prefix-based index in combination with the length filter can be considered state of the art given an efficient verification procedure (which we use).

Figure 6 shows the asymmetric prefix index for our running example, $\epsilon = 3$, $\pi^i = 2$. We probe $r_8 = \{3, 4, 7, 8\}$ and look up the lists of the tokens 3, 4, 7, and 8 (the length of the probing prefix is $\pi = 4$). Since we are only interested in the lookahead neighbors, i.e., all neighbors that follow r_8 in processing order, we need to inspect the lists only starting from the point where r_8 or a set $r_i \succ r_8$ appears. The length filter does not prune any candidate in this example, and the candidate set is $\{r_9\}$. Since $H(r_8, r_9) > \epsilon$, the lookahead neighborhood of r_8 is empty, $N_\epsilon^\succ(r_8) = \emptyset$.

In the context of the self-join, r_8 will be retrieved as a lookahead neighbor of r_6, which is processed before r_8. Therefore, the neighborhood list of r_6 will store r_8 and vice versa.

Join-Clust produces fewer candidates than Sym-Clust and is therefore faster. However, the efficiency of Join-Clust comes at the cost of a larger memory footprint since all neighborhoods must be materialized.

Complexity Analysis. A neighborhood query is a constant-time lookup and a traversal of $O(|N_\epsilon(r)|)$ neighbors. In the worst case, the join reports $O(n^2)$ pairs. Consequently, materializing the neighborhoods takes $O(n^2)$ time and space for $n = |R|$. The asymmetric prefix index requires only $O(n)$ space and does not dominate memory usage.

4 THE SPREAD ALGORITHM

We present *Spread*, a novel time- and space-efficient solution for the density-based clustering problem. Spread leverages the effective asymmetric index and clusters all sets by traversing the sets in processing order. We identify key challenges that must be solved, discuss the algorithm, prove its correctness, analyze time and space complexity, and sketch a multi-core extension.

4.1 Key Challenges

Since Spread uses an asymmetric neighborhood index, a processing order, $\succ$, must be imposed on the data points, and an index lookup of query point r retrieves only the lookahead neighbors, $N_\epsilon^\succ(r)$. To achieve a correct clustering without materializing the neighborhoods, three key challenges must be solved.

In the following discussion, we assume that all sets of R are processed in processing order. When the *current set* r_i is to be processed, we know the core status of all preceding sets, $r_j \prec r_i$, but we do not know the core status of any unprocessed sets, $r_k \succ r_i$. We further assume that all sets $r \in R$ that are directly density-reachable from any r_j that precedes r_i (i.e., are neighbors of a core point $r_j \prec r_i$) are assigned to the same cluster as the core point r_j; this may also include sets $r_k \succ r_i$ that have not been processed yet.

Core Status. A set r_i is core if $|N_\epsilon(r_i)| \geq$ minPts. Sym-Clust and Join-Clust have access to the full neighborhood, $N_\epsilon(r_i)$, thus deciding the core status of r_i is trivial. In contrast, Spread sees only the lookahead neighbors, $N_\epsilon^\succ(r_i)$. To identify the core status of r_i, however, additional knowledge about the size of the preceding neighborhood, $N_\epsilon^\prec(r_i)$, is required.

Consider probing $r_i = r_5$ in our running example. According to our assumptions, the core status of sets r_1–r_4 is known (only r_3 is core), and all neighbors of r_3 are in cluster $C_3 = \{r_1, r_3, r_4, r_6, r_7\}$. An index lookup of r_5 returns $N_\epsilon^\succ(r_5) = \{r_9, r_{10}\}$. Since $\left|N_\epsilon^\succ(r_5)\right| + 1 = 3 < 4 =$ minPts we cannot decide if r_5 is core. In fact, r_5 should be classified core since the full neighborhood is $N_\epsilon(r_5) = \{r_2, r_5, r_9, r_{10}\}$ (cf. red circle in Figure 3).

Border vs. Noise. Assume that the current set r_i is a non-core point that is not assigned to any cluster. We need to decide if r_i is border or noise. A border point has at least one core point in its neighborhood. None of the preceding neighbors, $r_j \in N_\epsilon^\prec(r_i)$, is core, otherwise r_i would be assigned to the cluster of r_j. Thus, r_i is core iff one of the lookahead neighbors is core. Unfortunately, we do not know the core status of the lookahead neighbors and can therefore not label r_i as border or noise.

Assume a core point, $r_k \in N_\epsilon^\succ(r_i)$, among the lookahead neighbors of r_i. When r_k is processed, r_k will not see r_i in its lookahead neighborhood since $r_i \prec r_k$. Therefore, r_i will not be included into the cluster of r_k and will wrongly be classified noise. The challenge is to correctly decide the border status of r_i despite seeing only the lookahead neighbors of r_i and r_k.

In our running example, r_1 is processed first. Thus, no core points are known and no clusters exist. $N_\epsilon^\succ(r_1) = \{r_3\}$ and r_1 remains noise (cf. blue circle in Figure 3). When the neighbor r_3 of r_1 is processed, r_3 will be detected as a core point and start a new cluster. However, since r_3 only sees its lookahead neighbors, $N_\epsilon^\succ(r_3) = \{r_4, r_6, r_7\}$, r_1 is not included into the cluster and is not detected as a border point.

Disconnected Clusters. Assume that the current set r_i is core and there is a core point $r_j \prec r_i$ in a cluster C_j, $r_i \notin C_j$. The current set r_i will assign all its lookahead neighbors to its cluster, $C_i = C_i \cup N_\epsilon^\succ(r_i)$ (C_i can be a new cluster started by r_i or an existing cluster to which r_i belongs). Unfortunately, we cannot assume that C_i and C_j are indeed distinct clusters: there can be a core point $r_k \succ r_i$ that density-reaches both r_i and r_j, i.e., C_i and C_j should form a single cluster. In general, multiple subclusters of the same DBSCAN cluster may grow independently. The challenge is to identify subclusters that should be merged and to merge them efficiently.

We process the current set $r_i = r_6$ in our running example. According to our assumptions, we know that r_3 and r_5 are core and we are aware of two clusters, $C_3 = \{r_1, r_3, r_4, r_6, r_7\}$, $C_5 = \{r_2, r_5, r_9, r_{10}\}$. In addition, assume that we know that r_6 is core. Then, r_6 extends its current cluster, C_3, with its lookahead neighbors $N_\epsilon^\succ(r_6) = \{r_8, r_{10}\}$. Note that r_{10} is already part of cluster C_5. Since we do not know the core status of r_{10}, we cannot decide if C_5 and C_3 should be merged into a single cluster. If r_{10} is core, r_5 and r_3 are density-reachable from r_{10} and should be in the same cluster. If r_{10} is a border point, however, the clusters must not be merged, and r_{10} can be assigned to either C_5 or C_3.

4.2 Data Structures

Disjoint-Set. The disjoint-set (or union-find) data structure maintains a dynamic collection of non-overlapping sets for n objects in $O(n)$ space [10, 41]. A typical use case is the efficient computation of (minimum) spanning trees. It supports three operations: (1) For a given element u, make_set(u) creates a new (singleton) set that contains u. (2) The union(u, v) operation merges the two sets that contain u resp. v into a new set. (3) find_set(u) returns the representative for the set that contains u or ∞ if u is not found. The amortized worst-case time complexity is $\Theta(\alpha(n))$ for all operations, $\alpha(.)$ being the inverse Ackermann function. In practice, $\alpha(n)$ is considered a constant. In our setting, set elements are subclusters, and the disjoint-set data structure links subclusters that belong to the same DBSCAN cluster.

Backlinks. The backlinks data structure of a set $r \in R$ is a collection of unique references to other sets s that precede r, $s \prec r$. The backlinks bl support the add operation, $bl \cup \{s\}$, which adds a reference to a new set s in time $O(1)$ (on average). Depending on the type of sets that are referenced in the backlinks, we distinguish core and non-core backlinks, denoted *c_bl* and *nc_bl*, respectively. We implement backlinks as unordered sets of integer identifiers.

4.3 The Algorithm

Algorithm 5 shows the pseudocode of Spread. We use the following notation: r is the current probing set, $s \succ r$ is a lookahead neighbor, and $x \prec r$ is a preceding neighbor. Initially, all sets are noise, i.e., their cluster identifier is $-\infty$, $\forall r \in R : r.cid = -\infty$. Although we initialize all sets in Algorithm 5 explicitly (lines 3–4), this can also be done during indexing (cf. Algorithm 2).

Algorithm Outline. Spread proceeds in three main steps: (1) A counter and the processing order guarantee that the cardinality of the ϵ-neighborhood is known when a set is processed despite using the asymmetric prefix index. (2) Each set is assigned to a subcluster solely based on its lookahead neighboorhood. Subclusters of the same DBSCAN cluster are linked in a subcluster graph. Backlinks ensure that we do not miss border sets or links between subclusters. (3) Each connected component in the subcluster graph represents a DBSCAN cluster.

Core Status. A set r is core if $|N_\epsilon(r)| \geq$ minPts. In Spread, however, only $N_\epsilon^\succ(r)$ is computed. To capture the cardinality of $N_\epsilon^\prec(r)$, we store a density counter with each set r, denoted $r.dens$. Initially, $\forall r \in R : r.dens = 1$. For every lookahead neighbor $s \in N_\epsilon^\succ(r)$, $r.dens$ and $s.dens$ are incremented (due to the symmetry of the distance). Core set identification is highlighted in green.

Border vs. Noise. A probing set r that is not core is a border set iff $\exists y \in N_\epsilon(r) : y$ is core. Due to our processing order and the fact that only $N_\epsilon^\succ(r)$ is computed, the existence of a core neighbor y may be unknown when r is probed. However, for each $s \in N_\epsilon^\succ(r)$, we know that r is part of $N_\epsilon^\prec(s)$. We store this information by adding r to the non-core backlinks $nc\_bl[s]$ of each $s \in N_\epsilon^\succ(r)$ (lines 31–33). Then, the first $s \in N_\epsilon^\succ(r)$ that becomes core claims r (and all other unassigned sets in $nc\_bl[s]$) as border point for its subcluster. If none of the neighbors $s \in N_\epsilon^\succ(r)$ becomes core, then r remains noise. Lines 26–30 deal with a special case: If any $s \in N_\epsilon^\succ(r)$ is already core when r is probed, then s claims r immediately without adding r to its non-core backlinks. The relevant code lines are marked in red.

Subcluster Linkage. If the probing set r is core and a core neighbor y is part of another subcluster, the subclusters of r and y must be linked in our subcluster graph. The subcluster graph represents all connected components of subclusters, each of which is

a DBSCAN cluster. We use the disjoint-set data structure ds to track the connected components. Two subclusters u, v are linked by $ds.\text{union}(u, v)$. We may not be able to determine if there is a set $s \in N_\epsilon^\succ(r)$ that is core before s is probed. We use the core backlinks, c_bl, to book-keep potential subclusters for linkage: r adds its subcluster identifier to $c\_bl[s]$ of each $s \in N_\epsilon^\succ(r)$ (lines 22-23). After $N_\epsilon^\succ(r)$ has been processed, a link between the subcluster of r and every entry in $c\_bl[r]$ is created (line 24). The special case when s is already core allows us to create the link immediately without using core backlinks (lines 20–21). Linkage is only required if two subclusters coalesce (condition in line 19). Otherwise, r simply claims $s \in N_\epsilon^\succ(r)$ for its subcluster (lines 17–18). Linkage of subclusters is highlighted in blue ▢.

All backlinks of r are released after r has been processed to save memory (line 34). The subcluster graph in ds is used to assign consistent cluster IDs in a final scan over R (lines 35–36).

4.4 Correctness

We show that Algorithm 5 partitions R into DBSCAN clusters (cf. Definition 2.1). Set $r_i \in R$ is the i-th set of R in processing order. We prove the correctness by induction over i and increasing subsets $R^i \subseteq R$. $R^0 = \emptyset$, $R^i = R^{i-1} \cup \{r_i\} \cup N_\epsilon^\succ(r_i)$ for $1 \leq i \leq n = |R|$, thus $R^n = R$. Due to space constraints, we omit the full proofs and only provide the invariants that must be shown.

Core Status. The core status of set r_i is determined in the i-th iteration of the main loop. r_i is core if $\text{minPts} \leq |N_\epsilon(r_i)| = 1 + \left|N_\epsilon^\prec(r_i)\right| + \left|N_\epsilon^\succ(r_i)\right|$. In line 5, $r_i.dens = 1 + \left|N_\epsilon^\prec(r_i)\right|$. Lines 6–11 compute $N_\epsilon^\succ(r_i)$. The index lookup in line 6 returns candidate set M, $N_\epsilon^\succ(r_i) \subseteq M \subseteq \{s \mid s \succ r_i\}$. Every set $s \in M$ is verified in line 9 such that $N_\epsilon^\succ(r_i)$ is available starting from line 12.

Lemma 4.1. *Algorithm 5 correctly identifies* all *core sets in R.*

Proof Sketch. We show that at the start of the i-th iteration in line 5, for all r_k and r_j, $1 \leq k < i \leq j$ the following invariants hold: (I1) $r_k.dens = |N_\epsilon(r_k)|$; (I2) $r_j.dens = 1 + \left|\{r_k \mid r_j \in N_\epsilon^\succ(r_k)\}\right|$, i.e., $r_i.dens = 1 + \left|N_\epsilon^\prec(r_i)\right|$. Further, (I3) in line 12 of the i-th iteration, $r_i.dens = |N_\epsilon(r_i)|$, i.e., Algorithm 5 correctly identifies the core status of r_i. □

Border vs. Noise. Lines 25–33 cover the case that r_i is not core. If any $s \in N_\epsilon^\succ(r_i)$ qualifies as core, s claims r_i. Otherwise, r_i is stored in the non-core backlinks $nc\_bl[s]$ of every $s \in N_\epsilon^\succ(r_i)$ (lines 31–33). The next core neighbor in processing order claims r_i (lines 14–15) such that all border sets are assigned to a cluster.

Lemma 4.2. *Algorithm 5 correctly clusters* all *border sets in R.*

Proof Sketch. At the start of the i-th iteration, the following invariant holds for all border sets $r_k \in \mathcal{B}$, $1 \leq k < i$: if r_k is not stored in $nc\_bl[s]$ for any $s \in N_\epsilon^\succ(r_k)$, $s = r_i$ or $s \succ r_i$, then r_k is assigned to the cluster of a core point in its neighborhood. □

Subcluster Linkage. Lines 12–24 cover the case that r_i is core. Each core point may form a subcluster on its own or together with other core points. We must ensure that all subclusters of the same DBSCAN cluster are linked in the disjoint-set, ds.

Lemma 4.3. *Algorithm 5 correctly links* all *subclusters in R.*

Proof Sketch. At the start of the i-th iteration, the following invariant holds for all core neighors $c \in CN(r_k) = N_\epsilon(r_k) \cap C$ of a core set $r_k \in C$, $1 \leq k < i$: (a) c and r_k have the same cluster representative (in ds), or (b) c is stored in some $c\_bl[s]$, $s \in N_\epsilon^\succ(r_k)$, $s = r_i$ or $s \succ r_i$. □

Algorithm 5: Spread(R, ϵ, minPts)

Input: Collection of sets R, distance threshold ϵ, min. density minPts
Result: A correct DBSCAN clustering of R w.r.t. ϵ, minPts

```
1  ds ← new disjoint-set; nc_bl, c_bl ← new backlinks;
2  I ← Create-Index(R, ε);
3  foreach r ∈ R do
4      r.dens ← 1; r.cid ← −∞; ds.make_set(r.id);
5  foreach r ∈ R in processing order do
6      M ← Probe(r, I, ε);
7      N_ε^≻(r) ← ∅;
8      foreach (s, po) ∈ M do // po ... prefix overlap
9          if Verify-Pair(r, s, ε, po) then
10             r.dens ← r.dens + 1; s.dens ← s.dens + 1;
11             N_ε^≻(r) ← N_ε^≻(r) ∪ {s};
12     if r.dens ≥ minPts then // r is core
13         if r.cid = −∞ then r.cid ← r.id;
14         foreach x ∈ nc_bl[r] do // claim border sets x ≺ r
15             if x.cid = −∞ then x.cid ← r.cid;
16         foreach s ∈ N_ε^≻(r) do // s ≻ r
17             if s.cid = −∞ then // claim unclaimed s ≻ r
18                 s.cid ← r.cid
19             else if r.cid ≠ s.cid then // s already claimed
20                 if s.dens ≥ minPts then // s is core
21                     ds.union(r.cid, s.cid) // link subclusters
22                 else // remember core neighbor r
23                     c_bl[s] ← c_bl[s] ∪ {r.cid}
24         foreach x ∈ c_bl[r] do ds.union(r.cid, x);
25     else // r is not core, i.e., r.dens < minPts
26         if r.cid = −∞ then // claim potential border set r
27             foreach s ∈ N_ε^≻(r) do
28                 if s.dens ≥ minPts then // s is core
29                     if s.cid = −∞ then s.cid ← s.id;
30                     r.cid ← s.cid; break;
31         if r.cid = −∞ then // remember potential border set r
32             foreach s ∈ N_ε^≻(r) do
33                 nc_bl[s] ← nc_bl[s] ∪ {r}
34     release c_bl[r] and nc_bl[r] // not needed anymore
35 foreach r ∈ R do // final assignment of cluster IDs
36     if r.cid ≠ −∞ then r.cid ← ds.find_set(r.cid);
```

Theorem 4.4. *Algorithm 5 returns a correct set clustering* $\Gamma = \{C_1, C_2, \ldots, C_k\}$ *of R according to Definition 2.1.*

Proof Sketch. By Lemmata 4.1–4.3 and due to our final scan over R (lines 35–36), $x.cid = ds.\text{find\_set}(x.cid)$ holds for all $x \in R$. Initially, $x.cid = -\infty$ for all $x \in R$. The cluster IDs are updated only for border and core sets. Consequently, $x.cid = -\infty$ holds for all $x \in R \setminus (C \cup \mathcal{B}) \equiv \mathcal{N}$, i.e., also noise is correctly identified. □

4.5 Complexity Analysis

Memory. The asymmetric prefix index requires $O(n)$ space. In addition, Spread maintains the following data structures. (i) A density counter for each set $r \in R$ requires $O(n)$ space. (ii) A disjoint-set data structure with at most $O(n)$ entries, i.e., the disjoint-set structure requires $O(n)$ space [41]. (iii) In the worst case, we allocate two backlink structures for each $r \in R$, i.e., $O(n)$ backlinks. We release $c\_bl[r]$ and $nc\_bl[r]$ after probing r. Backlinks are only extended in lines 23 and 33. However, both lines are executed iff $\nexists s \in N_\epsilon^>(r) : s$ is core. Set s is core iff $s.dens \geq$ minPts, and the density is updated for every neighbor, therefore any backlink holds at most minPts entries. As a result, no more than $O(n \cdot \text{minPts})$ entries are allocated, thus requiring $O(n)$ space since minPts and ϵ are constants. *Runtime.* For each $r \in R$, we process $O\left(\left|N_\epsilon^>(r)\right|\right)$ neighbors and the backlinks of r if it is core. Recall that the disjoint-set operations take constant time. Therefore, the final for-loop (lines 35–36) runs in $O(n)$ time. Overall, Spread runs in $O\left(n^2\right)$ time and $O(n)$ space.

4.6 Multi-core Extension

Spread is designed as a single-core algorithm. We sketch an extension to multi-core processors that requires little synchronization between threads. Our extension is based on the observation that Spread spends most of the runtime in neighborhood computations (lines 6-11). While for some datasets the neighborhood computation accounts for only about half of the overall runtime (e.g., 55% for ORKUT, $\epsilon = 3$), for the configuration with the highest runtime in our experiments (CELONIS1, $\epsilon = 5$), Spread spends over 99% of the runtime in computing the neighborhoods.

We distribute the workload to $k + 1$ threads, $T_1, T_2, \ldots, T_{k+1}$. Threads $T_1 - T_k$ are responsible for the neighborhood computations (lines 6-11), T_{k+1} performs the actual clustering (lines 12-34). The runtime of the other steps in the algorithm is negligible.

Neighborhood Computation. Let $r_i \in R$, $1 \leq i \leq |R|$ be the i-th set of R in processing order. Thread T_j, $1 \leq j \leq k$, computes the neighborhoods $N_\epsilon^>(r_i)$ of all r_i with $j = i \mod k$ (i.e., round robin). Each thread processes the assigned sets r_i in processing order (i.e., increasing values of i). The neighborhood computation in Algorithm 5 is interleaved with updating the density counters of r_i and its neighbors. Only this step requires synchronization (e.g., using atomic writes) since multiple threads may access the same counter concurrently. We do not expect congestions since the density updates are distributed over all neighbors.

Cluster Scan. Thread T_{k+1} scans the sets in processing order and performs the steps in lines 12-34 (maintain backlinks and disjoint-set, assign preliminary cluster IDs). After processing a set r_i, the memory for the neighbors of r_i is released.

Synchronization. We need to make sure that T_{k+1} processes set r_i only after r_i's neighbors have been computed. This can be achieved with a lock (implemented as condition variable[2]) on r_i that is held by T_j, $j \leq k$, until the neighborhood of r_i is computed. T_{k+1} needs to get the lock on r_i before processing it.

Memory. T_{k+1} releases the neighbors after processing them. If the parallel neighborhood computation is faster than T_{k+1}, the precomputed neighborhoods will fill up the memory. This is avoided with a shared counter that is incremented by $T_1 - T_k$ (when they process a new set r_i) and is decremented by T_{k+1} (after processing r_i). The neighborhood computation of r_i is postponed until the counter is below some threshold that bounds the number of concurrently materialized lookahead neighborhoods.

[2] A queue of threads waiting for a condition to become true.

Table 2: Characteristics of datasets.

Dataset	Coll. Size	Set Size		Univ. Size
		avg.	max.	
BMS-POS[4]	$3.2 \cdot 10^5$	9.3	164	$1.7 \cdot 10^3$
FLICKR[5]	$1.2 \cdot 10^6$	10.1	102	$8.1 \cdot 10^5$
KOSARAK[6]	$6.1 \cdot 10^5$	11.9	$2.5 \cdot 10^3$	$4.1 \cdot 10^4$
LIVEJ[7]	$3.1 \cdot 10^6$	36.4	300	$7.5 \cdot 10^6$
ORKUT[7]	$2.7 \cdot 10^6$	119.7	$4.0 \cdot 10^4$	$8.7 \cdot 10^6$
SPOT[8]	$4.4 \cdot 10^5$	12.8	$1.2 \cdot 10^4$	$7.6 \cdot 10^5$
CELONIS1	$8.2 \cdot 10^6$	20.3	91	$1.2 \cdot 10^4$
CELONIS2	$2.6 \cdot 10^6$	22.1	130	$3.5 \cdot 10^3$

5 EXPERIMENTAL EVALUATION

Algorithms. We compare our solution, Spread, against the two baseline approaches Sym-Clust and Join-Clust (cf. Section 3). All algorithms are single-threaded C++ implementations (2017 standard). Our implementations of Spread, Join-Clust, and the index of Sym-Clust follow the guidelines by Mann et al. [29], e.g., regarding symmetric and asymmetric prefix index, candidate generation, and optimized prefix-based verification.

Datasets. We execute all experiments on 13 real-world datasets: (a) Nine of the datasets where previously used for benchmarking set similarity joins [16, 29]: BMS-POS, DBLP, ENRON, FLICKR, KOSARAK, LIVEJ, NETFLIX, ORKUT, and SPOT. For a description of the datasets and preprocessing instructions[3] we refer to Mann et al. [29]. (b) Four large real-world datasets from the process mining domain, CELONIS1–4, that store one set per process. Compared to most datasets of the join benchmark, the universe size of these datasets is rather small. Table 2 summarizes important characteristics of our benchmark data.

Due to space constraints we omit detailed results for the following datasets: (a) DBLP, ENRON, and NETFLIX show very low runtimes (< 4s) and a small and stable memory footprint (< 1GiB) for all algorithms and configurations. (b) CELONIS3–4 show results similar to the other process mining datasets.

Parameters. The algorithms take two parameters: the neighborhood radius, ϵ, and the density, minPts. Typically, density-based clustering is sensitive to ϵ and quite robust to minPts. In our experiments, we vary both parameters: $\epsilon \in \{2, \mathbf{3}, 4, 5\}$ and minPts $\in \{2, 4, 8, \mathbf{16}, 32, 64, 128\}$ (defaults in bold font).

Environment. All experiments have been conducted on a 64-bit machine with 2 physical Intel Xeon E5-2630 v3 CPUs, 2.40GHz, 8 cores each (i.e., 16 logical processors, hyper-threading disabled). The cores share a 20MiB L3 cache and have another 256KiB of independent L2 cache. The system has 96GiB of RAM and runs Debian 10 Buster (Linux 4.19.0-12-amd64 #1 SMP Debian 4.19.152-1 (2020-10-18)). Our code is compiled with `clang`[9] version 7, highest optimization level (`-O3`). The runtime is measured with `clock_gettime`[10] at process level, memory usage is the `heap`

[3] http://ssjoin.dbresearch.uni-salzburg.at/datasets.html
[4] BMS-POS: http://www.kdd.org/kdd-cup/view/kdd-cup-2000 [52]
[5] FLICKR: Bouros et al. [6]
[6] KOSARAK: http://fimi.uantwerpen.be/data/
[7] LIVEJ, ORKUT: http://socialnetworks.mpi-sws.org/data-imc2007.html [30]
[8] SPOT: Pichl et al. [33]
[9] https://releases.llvm.org/7.0.0/tools/clang/docs/ReleaseNotes.html
[10] https://man7.org/linux/man-pages/man2/clock_gettime.2.html

Table 3: Index & cluster statistics for $\epsilon = 3$, minPts = 16.

(a) BMS-POS.

	Candidates	True Positives	Clusters
Sym-Clust	$3.9 \cdot 10^9$	$38.0 \cdot 10^6$	1
Join-Clust	$640.0 \cdot 10^6$	$38.0 \cdot 10^6$	1
Spread	$640.0 \cdot 10^6$	$38.0 \cdot 10^6$	1

(b) KOSARAK.

	Candidates	True Positives	Clusters
Sym-Clust	$40.7 \cdot 10^9$	$2.8 \cdot 10^9$	5
Join-Clust	$7.0 \cdot 10^9$	$2.8 \cdot 10^9$	5
Spread	$7.0 \cdot 10^9$	$2.8 \cdot 10^9$	5

(c) CELONIS1.

	Candidates	True Positives	Clusters
Sym-Clust	$644.6 \cdot 10^9$	$7.4 \cdot 10^6$	5,075
Join-Clust	$131.5 \cdot 10^9$	$7.4 \cdot 10^6$	5,075
Spread	$131.5 \cdot 10^9$	$7.4 \cdot 10^6$	5,075

peak of Linux `memusage`[11] (using `LD_PRELOAD`). A single instance is executed at a time with no other load on the machine.

5.1 Index & Cluster Statistics

We compare the number of candidates, true positives, and the number of clusters. The numbers are sums over all region queries. Table 3 shows the results obtained for BMS-POS, KOSARAK, and CELONIS1. We observe that Spread produces exactly the same number of candidates as Join-Clust since both solutions use the asymmetric index. Sym-Clust generates significantly more candidates due to the symmetric prefix index and the symmetric distance computations. For CELONIS1, Spread and Join-Clust verify about 5 times fewer candidates compared to Sym-Clust.

5.2 Runtime Efficiency

We measure the overall runtime, i.e., the CPU time that is required to cluster all sets into DBSCAN clusters (excluding the time to load the data from disk). Figure 7 shows the results for varying ϵ (minPts = 16). We observe that Sym-Clust is not competitive in terms of overall runtime in most cases. For all datasets, except KOSARAK and SPOT, the runtime of Sym-Clust increases much faster with ϵ than observed for Join-Clust and Spread. This is mainly due to the use of the symmetric prefix index (more candidates) and redundant computations (symmetric pairs).

Our experiments reveal that Join-Clust suffers from the following issues: (i) High runtimes for LIVEJ, ORKUT, and SPOT due to the expensive neighborhood materialization. (ii) Join-Clust runs out of memory for many instances (missing points in plots), in particular for FLICKR (any ϵ), KOSARAK ($\epsilon \geq 4$), LIVEJ, ORKUT, and SPOT ($\epsilon \geq 3$).

Spread outperforms its competitors in most settings and is competitive with Join-Clust otherwise (cf. Figures 7a, 7g, and 7h). For CELONIS1 and CELONIS2, Spread outperforms Sym-Clust by almost an order of magnitude and is competitive with Join-Clust.

Figure 8 shows the runtime results for varying minPts values ($\epsilon = 3$). We observe that the runtime of all three solutions is quite robust to minPts. The insights are similar for all datasets and values of ϵ. We include the plots for BMS-POS and KOSARAK.

[11] https://man7.org/linux/man-pages/man1/memusage.1.html

5.3 Memory Efficiency

We study the memory usage of Join-Clust, Sym-Clust, and Spread. All three solutions store (i) the collection, (ii) the inverted index, (iii) the candidates, and (iv) the result of a region query on the heap. The symmetric prefix index of Sym-Clust is larger than the asymmetric index, but still linear in the collection size. Sym-Clust generates more candidates than Join-Clust and Spread (cf. Section 5.1), which both use the asymmetric prefix index . Join-Clust materializes all neighborhoods in main memory. Sym-Clust and Spread materialize only a single neighborhood at a time. Spread stores also backlinks and the disjoint-set in main memory.

Figure 11 shows our results for varying ϵ (minPts = 16, y-axis log scale). Join-Clust runs out of memory for many instances (cf. Section 5.2). The neighborhood materialization in Join-Clust can be memory intensive even for small values of ϵ. We observe different growth rates with increasing radius ϵ, which we attribute to the different neighborhood sizes. The memory consumption of Sym-Clust is significantly lower and robust to varying ϵ. Spread shows a similar behavior. In some cases (e.g., LIVEJ, ORKUT), Spread occupies even less memory than Sym-Clust. When few backlinks are materialized, the smaller asymmetric prefix index of Spread outweighs the storage overhead for the backlinks.

Figure 12 shows the memory usage over minPts ($\epsilon = 3$, log-log scale). The memory consumption of Sym-Clust and Join-Clust is stable w.r.t. increasing values of minPts, while the memory usage of Spread slightly increases. This is due to the number of concurrently stored backlinks: the larger minPts, the higher the chance that a succeeding core neighbor is not yet classified, which triggers the creation of a backlink entry. The memory grows slowly with increasing minPts and does not limit the scalability of Spread. We include the results for BMS-POS and KOSARAK, $\epsilon = 3$; other datasets and ϵ values show similar results.

Backlinks Peak. We evaluate the effect of releasing the backlinks of a set in Spread after the set has been processed (cf. line 34, Algorithm 5). Figures 10 and 14 show the peak number of allocated backlinks relative to the maximum number of backlinks for varying ϵ (minPts = 16) and minPts ($\epsilon = 3$), respectively. Since two backlink structures, core (green) and non-core (orange), are maintained for each set in R, at most $2|R|$ backlinks can be allocated (light blue). Deallocating the backlinks of probed sets is highly effective: Only a small fraction of the maximum number of backlinks is allocated at any point in time. For increasing values of ϵ and minPts also the number of allocated backlinks grows.

5.4 Scalability

We evaluate the scalability of Spread and its competitors to increasing data sizes. To this end, we increase the size of BMS-POS and KOSARAK using the procedure of Vernica et al. [42]. This approach does not affect the token universe, and the number of similar pairs in the dataset increases linearly with the data size.

Figure 9 (runtime) and Figure 13 (main memory) report the results for our default parameter setting. Spread shows runtimes similar to Join-Clust and outperforms Sym-Clust by a factor of about 12 (BMS-POS) resp. 5.7 (KOSARAK) on the largest dataset (×16). As we increase BMS-POS by a factor of 16, the runtime increases by a factor of 195 for Spread, 204 for Join-Clust, and 460 for SymClust. The memory grows linearly for all measured data points and increases by a factor of about 2 when we double the data size. Join-Clust requires 18-25 (BMS-POS) resp. 499-569 (KOSARAK) times more memory than its competitors and runs out of memory on KOSARAK except for the ×1 dataset.

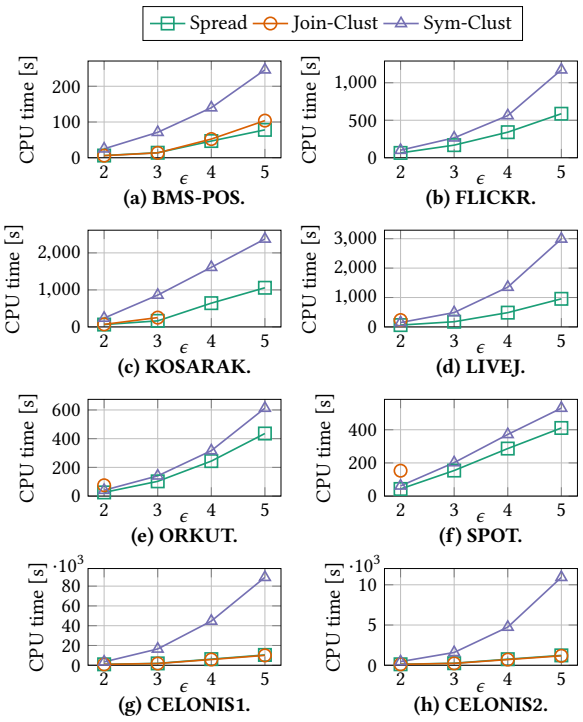


Figure 7: Runtime over ϵ, minPts = 16.

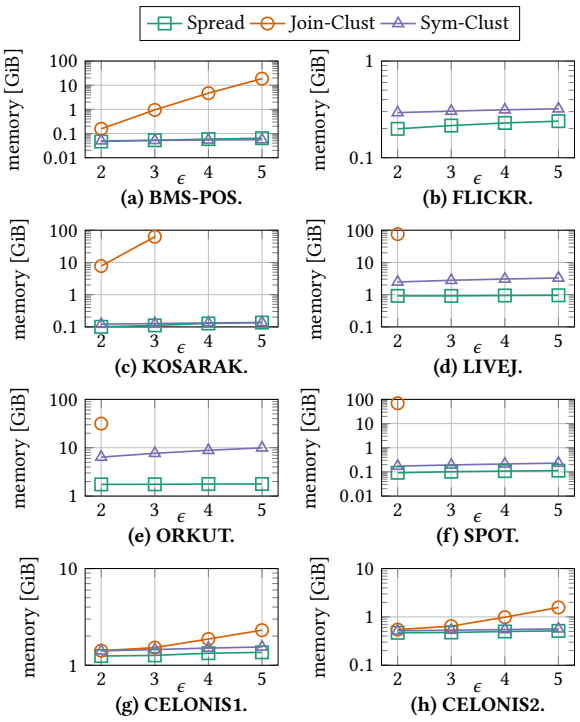


Figure 11: Main memory over ϵ, minPts = 16.

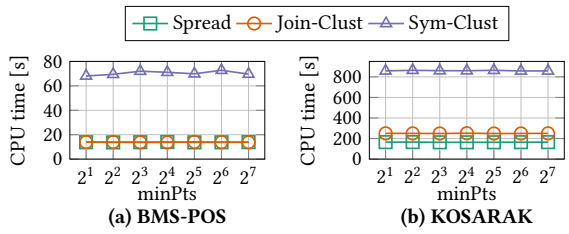


Figure 8: Runtime over minPts, ϵ = 3.

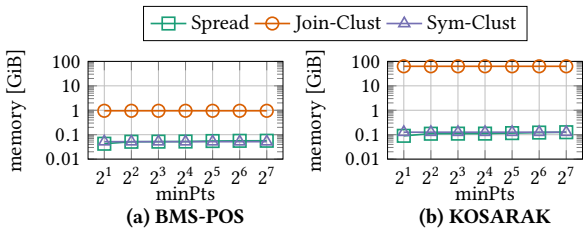


Figure 12: Main memory over minPts, ϵ = 3.

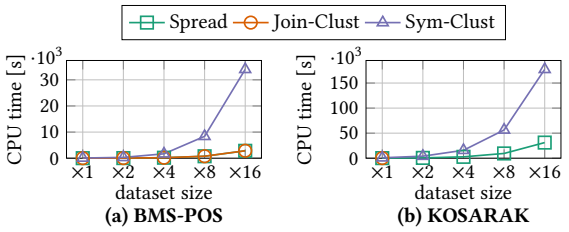


Figure 9: Runtime over data size, ϵ = 3, minPts = 16.

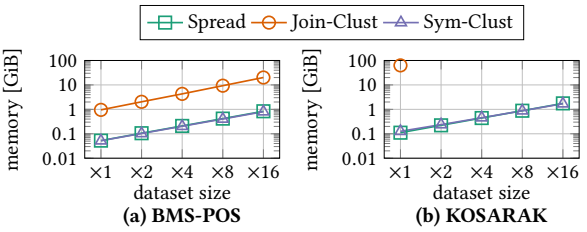


Figure 13: Main memory over data size, ϵ = 3, minPts = 16.

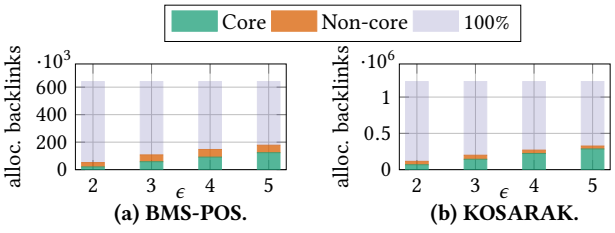


Figure 10: Backlinks peak over ϵ, minPts = 16.

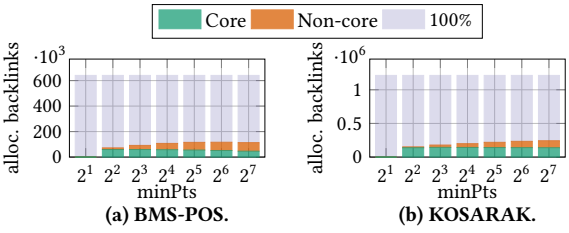


Figure 14: Backlinks peak over minPts, ϵ = 3.

Summarizing, Spread clearly outperforms Sym-Clust in runtime (by a factor of 5-12) and Join-Clust in memory usage (by more than an order of magnitude) as we increase the data size.

6 RELATED WORK

Indexes for Sets. Most set similarity joins operate on an inverted list index that maps signatures to candidate sets. Various signatures have been proposed [2, 8, 40, 45]. Prefixes [8] in conjunction with the length filter [1] have been shown to prune sets effectively. More sophisticated filters include positional and suffix filter [49], the removal filter [35], the position-enhanced length filter [28], and the adaptive prefix filter [44]. Wang et al. [46] leverage the similarity of the sets in an ϵ-neighborhood to reduce the overall number of false positives. Dong et al. [13] propose a size-aware algorithm that runs in $o(n^2) + O(k)$ time for k result pairs. Qin and Xiao [34] propose the pigeonring, a generalization of the pigeonhole principle that yields stronger constraints. Indexing and join techniques for sets have been studied extensively in the single-machine [29] and the distributed context [16].

Most of these approaches focus on self-joins, which order the sets and compute the lookahead neighborhood to avoid symmetric distance computations. In our work, we use the prefix filter, but any of the other asymmetric indexes is applicable.

Efficient Region Queries. Ester et al. [15] propose the first exact DBSCAN algorithm with $O(n \log n)$ runtime for vectors of arbitrary dimension. $O(n \log n)$ runtime holds for a small number of neighbors (compared to n) and an index with $O(\log n)$ lookup time. Henceforth, efficient region query computation has been of great interest and many improvements have been proposed. Brecheisen et al. [7] use minPts-nearest neighbor queries to identify core points and postpone the other distance computations until the distances are required to get a correct DBSCAN clustering. The proposed *Xseedlist* data structure is designed for expensive distance functions and assumes a cheap but selective filter. These assumptions do not hold for sets: The verification (i.e., distance computation) of candidate pairs has shown to be highly efficient [29] (a small number of integer comparisons). Brecheisen et al. must insert the candidates into the Xseedlist data structure, which maintains sorted lists of candidates. Due to the expensive sorting procedure, we do not expect Xseedlist to improve the DBSCAN algorithm for sets. TI-DBSCAN [25] exploits the triangle inequality to reduce the search space of region queries. The solution is not index-based, sorts the points w.r.t. a reference point, and shifts a window of size 2ϵ over the sorted points. The reference point is the point with minimal values in all dimensions. This is equivalent to the empty set, and our processing order in combination with the prefix index for sets subsumes this technique. Patwary et al. [32] introduce PARDICLE, a parallel approximate density-based clustering algorithm for Euclidean space. Its aim is to reduce the neighborhood computation time by sampling high-density regions. Kumar and Reddy [26] propose a new graph-based index structure called Groups. It discovers groups of patterns in two scans over the dataset and applies a standard DBSCAN afterwards. Groups accelerates region queries by pruning noise points effectively. This technique assumes Euclidean distance and does not consider Hamming distance or other set similarity measures. Recently, Jiang et al. [24] proposed SNG-DBSCAN, which prevents the computation of the full ϵ-neighborhood graph via subsampling its edges. This results in $O(sn^2)$-time complexity with s being the sampling rate. Under certain distribution assumptions, SNG-DBSCAN has been shown to preserve the exact ϵ-neighborhood graph for $s \approx (\log n)/n$ with $O(n \log n)$ runtime.

DBSCAN Techniques. Yang et al. [51] propose the distributed DBSCAN-MS clustering algorithm for metric spaces. DBSCAN-MS uses pivots to map the data from metric space to vector space, where it is partitioned in order to be distributed. A local DBSCAN is then executed on each partition. Our solution does not rely on the metric properties of set distances, but uses specialized set indexes. However, our techniques may be leveraged in the context of DBSCAN-MS, where the data points are ordered by one of the dimensions for efficient neighborhood queries.

Patwary et al. [31] propose PDSDBSCAN, a parallel DBSCAN algorithm that uses the disjoint-set data structure to connect data points into clusters. We only insert links between subclusters into the disjoint-sets structure, while PDSDBSCAN inserts a link for each neighbor, rendering the number of required union operations a bottleneck for this approach.

Böhm et al. [3] use a block-nested loop join and buffer the join result to reduce the number of block accesses required to compute ϵ-neighborhoods. CUDA-DClust [4] is a GPU-based solution that splits clusters into chains that are expanded from different starting points in parallel. In order to merge chains into clusters, a quadratic-size bit matrix is used. We maintain only a linear number of links and leverage disjoint-sets to merge clusters. Incremental DBSCAN algorithms [14] deal with updates on an existing clustering. Similar to our approach, these techniques may need to merge clusters when new points are inserted. None of the above solutions supports asymmetric neighborhood indexes.

Numerous parallel and distributed algorithms [9, 11, 18–21, 23, 36, 38, 47, 50] as well as approximations [17, 27, 43, 48] have been proposed. We present an exact, single-core solution for sets.

7 CONCLUSION

In this paper, we have investigated clustering techniques for large collections of sets. Our work was motivated by an application in process mining that models processes as sets to assess their similarity. We have shown that the solutions that are currently available, Sym-Clust and Join-Clust, are not satisfying: Sym-Clust is slow since it cannot use effective asymmetric set indexes, while Join-Clust is infeasible for many settings due to its excessive memory usage. We introduced a novel, density-based clustering algorithm, Spread, that can process data points in any user-defined order and is therefore fit for the use with asymmetric indexes. Spread combines the best of both worlds: It uses the effective asymmetric index of Join-Clust, but like Sym-Clust does not need to materialize the neighborhoods. We introduced so-called backlinks to guarantee a correct DBSCAN clustering and showed the correctness of our approach. To the best of our knowledge, Spread is the first DBSCAN-compliant algorithm that uses an asymmetric index and runs in linear space.

Spread uses the index as a black box and works with any data type. Interesting future work includes evaluating the performance of Spread for vector data, where candidates are generated using a sliding window that is shifted along one dimension. The data points in the window are candidates, i.e., the window simulates an asymmetric index for Spread.

ACKNOWLEDGMENTS

We thank Alexander Miller, Mateusz Pawlik, Thomas Hütter, Manuel Widmoser, Manuel Kocher, Daniel Ulrich Schmitt, Konstantin Thiel, Daniel Grittner, Christian Böhm, and Claudia Plant

for valuable discussions, and Manuel Kocher for typesetting Figures 1 and 3. This work was partially supported by the Austrian Science Fund (FWF): P 29859.

REFERENCES

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-Similarity Joins. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*. 918–929.

[2] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling Up All Pairs Similarity Search. In *Proc. of the Int. Conf. on World Wide Web (WWW)*. 131–140.

[3] Christian Böhm, Bernhard Braunmüller, Markus Breunig, and Hans-Peter Kriegel. 2000. High Performance Clustering Based on the Similarity Join. In *Proc. of the ACM Int. Conf. on Information and Knowledge Management (CIKM)*. 298–305.

[4] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. 2009. Density-Based Clustering Using Graphics Processors. In *Proc. of the ACM Int. Conf. on Information and Knowledge Management (CIKM)*. 661–670.

[5] R. P. Jagadeesh Chandra Bose and Wil M. P. van der Aalst. 2010. Trace Clustering Based on Conserved Patterns: Towards Achieving Better Process Models. In *Business Process Management Workshops*. 170–181.

[6] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. 2012. Spatio-Textual Similarity Joins. *Proc. of the VLDB Endowment* 6, 1 (Nov. 2012), 1–12.

[7] S. Brecheisen, H. . Kriegel, and M. Pfeifle. 2004. Efficient Density-Based Clustering of Complex Objects. In *Proc. of the IEEE Int. Conf. on Data Mining*. 43–50.

[8] S. Chaudhuri, V. Ganti, and R. Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*. 5–5.

[9] I. Cordova and T. Moh. 2015. DBSCAN on Resilient Distributed Datasets. In *Proc. of the Int. Conf. on High Performance Computing Simulation (HPCS)*. 531–540.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.).

[11] B. Dai and I. Lin. 2012. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. In *Proc. of the IEEE Int. Conf. on Cloud Computing*. 59–66.

[12] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. 2015. An Efficient Partition Based Method for Exact Set Similarity Joins. *Proc. of the VLDB Endowment* 9, 4 (Dec. 2015), 360–371.

[13] Dong Deng, Yufei Tao, and Guoliang Li. 2018. Overlap Set Similarity Joins with Theoretical Guarantees. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 905–920.

[14] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. 1998. Incremental Clustering for Mining in a Data Warehousing Environment. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*. 323–333.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. of the ACM Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*. 226–231.

[16] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 2018. Set Similarity Joins on Mapreduce: An Experimental Survey. *Proc. of the VLDB Endowment* 11, 10 (June 2018), 1110–1122.

[17] Junhao Gan and Yufei Tao. 2015. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 519–530.

[18] Markus Götz, Christian Bodenstein, and Morris Riedel. 2015. HPDBSCAN: Highly Parallel DBSCAN. In *Proc. of the Workshop on Machine Learning in High-Performance Computing Environments*. Article 2, 10 pages.

[19] D. Han, A. Agrawal, W. Liao, and A. Choudhary. 2016. A Novel Scalable DBSCAN Algorithm with Spark. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1393–1402.

[20] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. 2014. MR-DBSCAN: A Scalable MapReduce-Based DBSCAN Algorithm for Heavily Skewed Data. *Frontiers of Computer Science* 8, 1 (2014), 83–99.

[21] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan. 2011. MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce. In *Proc. of the IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*. 473–480.

[22] B.F.A. Hompes, J.C.A.M. Buijs, W.M.P. van der Aalst, P.M. Dixit, and J. Buurman. 2015. Discovering Deviating Cases and Process Variants Using Trace Clustering. In *Benelux Conf. on Artificial Intelligence*.

[23] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. 2004. Scalable Density-Based Distributed Clustering. In *Knowledge Discovery in Databases (PKDD)*. 231–244.

[24] Heinrich Jiang, Jennifer Jang, and Jakub Łącki. 2020. Faster DBSCAN via subsampled similarity queries. *CoRR* (2020).

[25] Marzena Kryszkiewicz and Piotr Lasek. 2010. TI-DBSCAN: Clustering with DBSCAN by Means of the Triangle Inequality. In *Rough Sets and Current Trends in Computing (RSCTC)*. 60–69.

[26] K. Mahesh Kumar and A. Rama Mohan Reddy. 2016. A fast DBSCAN clustering algorithm by accelerating neighbor searching using Groups method. *Pattern Recognition* 58 (2016), 39 – 48.

[27] Yinghua Lv, Tinghuai Ma, Meili Tang, Jie Cao, Yuan Tian, Abdullah Al-Dhelaan, and Mznah Al-Rodhaan. 2016. An efficient and scalable density-based clustering algorithm for datasets with complex structures. *Neurocomputing* 171 (2016), 9 – 22.

[28] Willi Mann and Nikolaus Augsten. 2014. PEL: Position-Enhanced Length Filter for Set Similarity Joins. In *Proc. of the Workshop Grundlagen von Datenbanken (CEUR Workshop Proceedings)*, Vol. 1313. 89–94.

[29] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *Proc. of the VLDB Endowment* 9, 9 (2016), 636–647.

[30] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proc. of the ACM Int. Conf. on Internet Measurement (SIGCOMM)*. 29–42.

[31] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, and A. Choudhary. 2012. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[32] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey. 2014. Pardicle: Parallel Approximate Density-Based Clustering. In *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. 560–571.

[33] Martin Pichl, Eva Zangerle, and Günther Specht. 2014. Combining Spotify and Twitter Data for Generating a Recent and Public Dataset for Music Recommendation. In *Proc. of the Workshop Grundlagen von Datenbanken (CEUR Workshop Proceedings)*, Vol. 1313. 35–40.

[34] Jianbin Qin and Chuan Xiao. 2018. Pigeonring: A Principle for Faster Thresholded Similarity Search. *Proc. of the VLDB Endowment* 12, 1 (2018), 28–42.

[35] Leonardo Andrade Ribeiro and Theo Härder. 2011. Generalizing Prefix Filtering to Improve Set Similarity Joins. *Information Systems* 36, 1 (2011), 62–78.

[36] A. Sarma, P. Goyal, S. Kumari, A. Wani, J. S. Challa, S. Islam, and N. Goyal. 2019. μDBSCAN: An Exact Scalable DBSCAN Algorithm for Big Data Exploiting Spatial Locality. In *Proc. of the IEEE Int. Conf. on Cluster Computing (CLUSTER)*. 1–11.

[37] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Transactions on Database Systems* 42, 3 (2017), 21.

[38] Hwanjun Song and Jae-Gil Lee. 2018. RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 1173–1187.

[39] Minseok Song, Christian W. Günther, and Wil M. P. van der Aalst. 2009. Trace Clustering in Process Mining. In *Business Process Management Workshops*. 109–120.

[40] Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, and Zhifeng Bao. 2019. Balance-aware Distributed String Similarity-based Query Processing System. *Proc. of the VLDB Endowment* 12, 9 (2019), 961–974.

[41] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM* 22, 2 (1975), 215–225.

[42] Rares Vernica, Michael J. Carey, and Chen Li. 2010. Efficient Parallel Set-Similarity Joins Using MapReduce. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 495–506.

[43] P. Viswanath and V. Suresh Babu. 2009. Rough-DBSCAN: A fast hybrid density based clustering method for large data sets. *Pattern Recognition Letters* 30, 16 (2009), 1477 – 1488.

[44] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can We Beat the Prefix Filtering? An Adaptive Framework for Similarity Join and Search. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 85–96.

[45] Pei Wang, Chuan Xiao, Jianbin Qin, Wei Wang, Xiaoyang Zhang, and Yoshiharu Ishikawa. 2016. Local Similarity Search for Unstructured Text. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 1991–2005.

[46] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2017. Leveraging Set Relations in Exact Set Similarity Join. *Proc. of the VLDB Endowment* 10, 9 (May 2017), 925–936.

[47] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-Efficient and Practical Parallel DBSCAN. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2555–2571.

[48] Y. Wu, J. Guo, and X. Zhang. 2007. A Linear DBSCAN Algorithm Based on LSH. In *Proc. of the Int. Conf. on Machine Learning and Cybernetics*, Vol. 5. 2608–2614.

[49] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient Similarity Joins for Near-Duplicate Detection. *ACM Transactions on Database Systems* 36, 3 (2011), 41.

[50] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. 1999. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Mining and Knowledge Discovery* 3, 3 (1999), 263–290.

[51] K. Yang, Y. Gao, R. Ma, L. Chen, S. Wu, and G. Chen. 2019. DBSCAN-MS: Distributed Density-Based Clustering in Metric Spaces. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*. 1346–1357.

[52] Zijian Zheng, Ron Kohavi, and Llew Mason. 2001. Real World Performance of Association Rule Algorithms. In *Proc. of the ACM Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*. 401–406.

Assess Queries for Interactive Analysis of Data Cubes

Matteo Francia
DISI - University of Bologna
Bologna, Italy
m.francia@unibo.it

Matteo Golfarelli
DISI - University of Bologna
Bologna, Italy
matteo.golfarelli@unibo.it

Patrick Marcel
University of Tours
Blois, France
patrick.marcel@univ-tours.fr

Stefano Rizzi
DISI - University of Bologna
Bologna, Italy
stefano.rizzi@unibo.it

Panos Vassiliadis
University of Ioannina
Ioannina, Greece
pvassil@cs.uoi.gr

ABSTRACT

Assessment is the process of comparing the actual to the expected behavior of a business phenomenon and judging the outcome of the comparison. In this paper we propose assess, a novel querying operator that supports assessment based on the results of a query on a data cube. This operator requires (1) the specification of an OLAP query over a measure of a data cube, to define the target cube to be assessed; (2) the specification of a reference cube of comparison (benchmark), which represents the expected performance of the measure; (3) the specification of how to perform the comparison between the target cube and the benchmark, and (4) a labeling function that classifies the result of this comparison using a set of labels. After introducing an SQL-like syntax for our operator, we formally define its semantics in terms of a set of logical operators. To support the computation of assess we propose a basic plan as well as some optimization strategies, then we experimentally evaluate their performance using a prototype.

1 INTRODUCTION

Assume an analyst wants to *assess* the state of milk sales in France for 2019. She will have to issue a query against an OLAP server to obtain a cube, and then ask: "how good, normal, surprising, etc. is the situation I observe for this particular cube as compared to some reference data?". Assessment, as a process, is about comparing the actual to the expected behavior and judging, for instance through a *labeling*, the outcome of the comparison. Examples of how to assess the status of a cube (or of each single cell of a cube) include its comparison to:

(1) ...a predefined target goal for the sales, e.g., because of the existence of a predefined KPI (Key Performance Indicator);
(2) ...a predefined golden standard, acting as a reference benchmark (e.g., comparing French milk sales against the EU average) or, as an example in another domain, comparing a stock value to the S&P 500 index);
(3) ...sibling cells, i.e., cells describing a similar context and sharing some dimension values (i.e., compare sales for yogurt and ice-cream in Greece in 2019, or milk sales in Spain and Italy for 2019);
(4) ...the expected status of the cube as can be predicted from the past (e.g., compare actual milk sales in December 2018 with those that can be predicted from the sales of the previous six months).
(5) ...a new, derived measure produced via a function whose formula involves other measures (e.g., profit=storeSales-storeCost).

This kind of tabular data assessment is consistently reported as a frequent activity of data explorers [3, 12, 23] who often use SQL in combination with languages like Python and R. Noticeably, assessment is one of the user's intentions considered in the *Intentional Analytics Model* (IAM), which has been envisioned as a way to tightly couple OLAP and analytics [4, 21]. The IAM approach relies on two major cornerstones: (i) the user explores the data space by expressing her analysis *intentions* rather than by explicitly stating what data she needs, and (ii) in return she receives both multidimensional data and knowledge insights in the form of annotations of interesting subsets of data. Among the five intention operators proposed, assess is meant to judge a cube measure with reference to some baseline.

In this paper we adopt the OLAP-centered nature of the IAM and operate in the context of a traditional OLAP environment with cubes, dimensions, hierarchies, and measures. This allows us to take advantage of the neat logical-level schema structure of OLAP and focus on the essence of the paper, which is proposing an assess operator to complement the traditional OLAP roll-up's and drill-down's. The idea of how to perform an assessment for the measure values of a cube encompasses (a) the specification of another cube, called *benchmark*, that represents the expected or desirable performance of the measure; (b) the *comparison* of the measure under investigation to the benchmark measure (for instance via a simple mathematical difference); and (c) the characterization, or *labeling*, of the status of the original cell based on the result of the comparison.

Example 1.1. Given a SALES cube, the user's intention described above can be expressed with this statement:

```
with SALES
for year = '2019', product = 'milk'
by year, product
assess quantity against 1000
using ratio(quantity, 1000)
labels {[0, 0.9): bad, [0.9, 1.1]: acceptable, (1.1,inf): good}
```

Intuitively, the total quantity of milk sold in France in 2019 is labeled as bad/acceptable/good depending on the ratio with the target value 1000. □

Summary of contributions. Our contributions can be listed as follows:

- We introduce a novel operator, assess, that allows to automatically evaluate and characterize the result of a cube query.

- We introduce alternatives for specifying benchmarks, comparison, and labeling schemes against which the results of a cube query can be compared and evaluated. For each alternative we provide rigorous definitions and semantics, based on a set of logical operators, as well an SQL-like syntax for the specification of an assess statement.
- We discuss alternative plans for the execution of assess statements and experimentally evaluate them for their efficiency and scalability.

Roadmap. In Section 2 we formalize the involved concepts and give definitions. In Section 3 we explain how assessments are computed and introduce the alternatives of the assess operator, while in Section 4 we provide its syntax and semantics. In Section 5 we present alternative strategies for query execution and in Section 6 we experimentally evaluate them. In Section 7 we discuss the related work. Finally, in Section 8 we summarize our findings and discuss our future work.

2 FORMALITIES

To simplify the formalization, we will restrict to consider linear hierarchies.

Definition 2.1 (Hierarchy and Cube Schema). A *hierarchy* is a triple $h = (L, \geq, \geq)$ where:

(i) L is a set of categorical *levels*, each coupled with a *domain* of values (a.k.a. as *members*), $Dom(l)$;
(ii) $\geq$ is a *roll-up* total order of L; and
(iii) $\geq$ is a *part-of* partial order of $\bigcup_{l \in L} Dom(l)$.

The part-of partial order is such that, for each couple of levels l and l' such that $l \geq l'$, for each member $u \in Dom(l)$ there is exactly one member $u' \in Dom(l')$ such that $u \geq u'$.

A *cube schema* is a couple $C = (H, M)$ where:

(i) H is a set of hierarchies;
(ii) M is a tuple[1] of numerical measures, each coupled with one aggregation operator $op(m) \in \{\mathsf{sum}, \mathsf{avg}, \ldots\}$.

Example 2.2. As a working example we will use cube schema SALES $= (H, M)$, where

$$H = \{h_{\mathsf{Date}}, h_{\mathsf{Customer}}, h_{\mathsf{Product}}, h_{\mathsf{Store}}\},$$
$$M = \langle \mathsf{quantity}, \mathsf{storeSales}, \mathsf{storeCost} \rangle,$$
$$\mathsf{date} \geq \mathsf{month} \geq \mathsf{year},$$
$$\mathsf{customer} \geq \mathsf{gender},$$
$$\mathsf{product} \geq \mathsf{type} \geq \mathsf{category},$$
$$\mathsf{store} \geq \mathsf{city} \geq \mathsf{country}$$

and $op(\mathsf{quantity}) = op(\mathsf{storeSales}) = op(\mathsf{storeCost}) = \mathsf{sum}$. As to the part-of partial order we have, for instance, Fresh Fruit $\geq$ Fruit and 1997-04-15 $\geq$ 1997. □

Aggregation is the basic mechanism to query cubes, and it is captured by the following definition of group-by set. As normally done when working with the multidimensional model, if a hierarchy h does not appear in a group-by set it is implicitly assumed that a complete aggregation is done along h.

Definition 2.3 (Group-by Set and Coordinate). Given cube schema $C = (H, M)$, a *group-by set* of C is a tuple of levels, at most one from each hierarchy of H. The partial order induced on the set of all group-by sets of C by the roll-up orders of the hierarchies in H, is denoted with $\geq_H$. A *coordinate* of group-by set G is a tuple of members, one for each level of G. Given coordinate γ of group-by set G and another group-by set G' such that $G \geq_H G'$, we will denote with $rup_{G'}(\gamma)$ the coordinate of G' whose members are related to the corresponding members of γ in the part-of orders, and we will say that γ *roll-up*s to $rup_{G'}(\gamma)$. By definition, $rup_G(\gamma) = \gamma$.

Definition 2.4 (Detailed Cube). Let G_0 be the top group-by set in the $\geq_H$ partial order (i.e., the finest one). A *detailed cube* over C is a partial function C_0 that maps the coordinates of G_0 to a numerical value for each measure m in M.

The function is partial since cubes are normally *sparse*: not all possible business events actually occur, and a coordinate participates in the function only if the event it describes took place. Each coordinate γ that participates in C_0, with its associated tuple t of measure values, is called a *cell* of C_0 and denoted $c = \langle \gamma, t \rangle$. With a slight abuse of notation, we will also consider a cube as the set of the coordinates corresponding to its cells, so we will write $\gamma \in C_0$ to state that $\langle \gamma, t \rangle$ is a cell of C_0.

Example 2.5. Three group-by sets of SALES are

$$G_0 = \langle \mathsf{date}, \mathsf{customer}, \mathsf{product}, \mathsf{store}\} \rangle$$
$$G_1 = \langle \mathsf{date}, \mathsf{type}, \mathsf{country} \rangle$$
$$G_2 = \langle \mathsf{month}, \mathsf{category} \rangle$$

where $G_0 \geq_H G_1 \geq_H G_2$. G_0 is the top group-by set. G_1 aggregates sales by date, product type, and store country (for all customers), G_2 by month and category (for all customers and stores). Examples of coordinates of the three group-by sets are, respectively,

$$\gamma_0 = \langle \text{1997-04-15}, \text{Eric Long}, \text{Lemon}, \text{SmartMart} \rangle$$
$$\gamma_1 = \langle \text{1997-04-15}, \text{Fresh Fruit}, \text{Italy} \rangle$$
$$\gamma_2 = \langle \text{1997-04}, \text{Fruit} \rangle$$

where $rup_{G_1}(\gamma_0) = \gamma_1$ and $rup_{G_2}(\gamma_1) = \gamma_2$. An example of cell of a detailed cube over SALES is $\langle \gamma_0, \langle \mathsf{quantity} = 5, \mathsf{storeSales} = 20, \mathsf{storeCost} = 12 \rangle \rangle$. □

Definition 2.6 (Cube Query and Derived Cube). Given a detailed cube C_0 over schema C, a *query* over C_0 is a quadruple $q = (C_0, G_q, P_q, M_q)$ where:

(i) G_q is a group-by set of C;
(ii) P_q is a (possibly empty) set of selection predicates each expressed over one level of H;
(iii) $M_q \subseteq M$.

The result of q is called a *derived cube*, i.e., a partial function that assigns to each coordinate γ of G_q satisfying the conjunction of the predicates in P_q and to each measure m in M_q the value computed by applying $op(m)$ to the values of m for all the coordinates of C_0 that roll-up to γ, provided that such coordinates of C_0 exist.

Like detailed cubes, even derived cubes can be sparse; a coordinate γ does not participate in the function if there is no coordinate in C_0 that rolls-up to γ. Like for detailed cubes, we will write $\gamma \in C$ to state that γ is a coordinate of the derived cube C. Consistently with this, we will denote with $|C|$ the number of coordinates in C.

Example 2.7. A cube query over SALES is $q = (C_0, G_q, P_q, M_q)$ where $G_q = \langle \mathsf{product}, \mathsf{country} \rangle$, $P_q = \{\mathsf{type} = \text{'Fresh Fruit'}, \mathsf{country} = \text{'Italy'}\}$, and $M_q = \langle \mathsf{quantity} \rangle$. A cell of the resulting

[1] When dealing with tuples we will write $t_1 = t_{2|_{sort(t_1)}}$ to denote that tuple t_1 is contained in tuple t_2; (t_1, t_2) to denote the tuple that concatenates t_1 and t_2; $t_{|_X}$ to denote the projection of tuple t on its component(s) X [1].

```
select country, product, sum(quantity) as quantity
from sales s
  join customer c on c.ckey = s.ckey
  join product p on p.pkey = s.pkey
where type = 'Fresh Fruit' and country = 'Italy'
group by country, product
```

Listing 1: Getting the sales of fresh fruit products in Italy (Example 2.7)

cube is $\langle\langle \text{Apple}, \text{Italy}\rangle, \langle \text{quantity} = 100\rangle\rangle$. The SQL formulation of q on a star schema is given in Listing 1. □

3 COMPUTING AN ASSESSMENT

Basically, the assessment of the values of a measure m in a cube C (called *target cube*) is done in three steps:

(1) the specification of a *benchmark*, i.e., a cube B such that (i) its cells can be mapped one-to-one with the cells of C, and (ii) it has a measure m' representing the expected/acceptable/normal performance of m;
(2) the cell-wise *comparison* of m to m', which can be done in a basic way (e.g., algebraic/absolute/normalized difference, percentage) or using more elaborate schemes (e.g., z-scoring), possibly after applying some *transformations* to m and m' (e.g., to compute derived measures);
(3) the characterization, or *labeling*, of the status of each cell of C based on the result of the comparison; in the simplest case, this is done using a set of rules that map the result of the comparison to a set of predefined labels (e.g., "insufficient", "excellent", etc.).

3.1 Benchmarks

The specification of the benchmark is given by the analyst at the posing of the query. Thus, the question is *"tell me how we are doing with respect to this benchmark"*.

A thorough comparison of a target cube C against a benchmark B would require that the latter comes with the same level members so that, for each cell of C, we can map onto a cell of B. However, in practical cases, due to cube sparsity, there is no guarantee that all cells can be mapped —especially if the benchmark is retrieved from the web or other external data sources. Thus, in the following we provide a broad definition of the conditions under which two cubes are joinable, i.e., one of them can be used as a benchmark to assess the other; in this definition, we will just require that the two cubes have the same group-by set.

Definition 3.1 (Cube Joinability). Let a target cube C over cube schema $\mathcal{C}$ and a benchmark B over $\mathcal{B}$ (where possibly, but not necessarily, $\mathcal{B} = \mathcal{C}$) be given. Let $q = (C_0, G_C, P_C, M_C)$ and $q' = (B_0, G_B, P_B, M_B)$ be the queries that resulted in C and B, respectively. We say that C and B are *joinable* if

$$G_C = G_B$$

In OLAP terms, two cubes are joinable if a drill-across is possible between C and B.

Let $\mathcal{C} = (H, M)$ be the schema of the target cube C, and C_0 be the detailed cube from which C is derived. There are four types of benchmarks we consider in our approach:

- *Constant benchmarks.* Here the user simply wants to assess the cells of the target cube C against some fixed value, as typically done with key performance indicators. In this case, the benchmark B has schema $\mathcal{B} = (H, \langle m_{const}\rangle)$; its cells have exactly the same coordinates as C, and all of them store a constant value in m_{const}. The cell-to-cell mapping is trivially based on equality of coordinates.
- *External benchmarks.* Here the user's goal is to assess the target cube against the data stored in a cube with schema $\mathcal{B} = (H', M')$. In principle, as long as $\mathcal{B}$ includes the group-by of the target cube (which ensures joinability), it is not necessary to impose further constraints on $\mathcal{B}$. However, for simplicity, in the following we will assume that the external benchmark has been reconciled with the target cube so that $H = H'$ and that all necessary transcodings to level members have been applied (see e.g. [10] for an approach that can be pursued to this end). Thus, also in this case, mapping is based on equality of coordinates.
- *Sibling benchmarks.* The idea here is to compare the values of a measure in a slice on member $u \in Dom(l)$ with the values of the same measure in another slice of C related to a sibling member $u_{sib} \in Dom(l)$ (e.g., assess the sales of fruit in Italy with reference to those in France). In this case, the benchmark has the same schema $\mathcal{C}$ of the target cube. Both cubes have the same group-by set, but while the cells in C are those obtained from C_0 using predicate $l = u$, those in B are obtained from C_0 using predicate $l = u_{sib}$. Then the cell-to-cell mapping is established by replacing u with u_{sib} in each coordinate of C.
- *Past benchmarks.* In this case the user wants to assess the values taken by a measure m in some time slice with the values that can be predicted for m based on a number of past time slices. Like in the previous case, it is $\mathcal{B} = \mathcal{C}$. The cells of B have exactly the same coordinates as C, but the (actual) values of m are replaced with the predicted ones.

Example 3.2. Let C be the derived cube obtained by query q in Example 2.7 (total quantity sold by product and country for fresh fruit products and Italy). An example of (joinable) sibling benchmark is B returned by q', being q' obtained from q by replacing Italy with France. B can be used to assess the sales of fresh fruit in Italy against those in France. The cell-to-cell mapping is established by replacing Italy with France; so, for instance, coordinate $\langle \text{Apple}, \text{Italy}\rangle$ is mapped onto $\langle \text{Apple}, \text{France}\rangle$. □

3.2 Comparison & Transformation

The essence of assessment is to contrast the actual performance against its expected value. Thus, the goal of this step is to provide the means to express and perform the evaluation of how far apart the query result and the benchmark are. We refer to this action as *comparison* to express the idea that this is not necessarily a simple measure difference. Modeling-wise, we assume that a library of comparison functions, all with signature $\delta : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, is available to the users. Practically, a cell-wise comparison between measures of the target and benchmark can be easily implemented via different functions obeying the above signature, the simplest choice being a *difference* (either *algebraic*, or *absolute*, or *normalized*, etc.). In our examples, we will use two library functions of our system, named difference and ratio.

One could possibly expect that, once the target cube and the benchmark have been obtained, their comparison is immediately applicable. Interestingly, this is not always the case, since the comparison may require the computation of derived measures. For instance, with reference to the SALES cube, comparing the actual profit for some given sales requires to compute a derived measure as profit = storeSales – storeCost. Clearly, this requires

```
def difference(a, b):
    return a - b

def minmaxnorm(a):
    minv = a.min()
    maxv = a.max()
    return (a - minv) / (maxv - minv)
```

Listing 2: Implementation of the difference and minmaxnorm functions

that either of, or both, the target and the benchmark measures pass through a set of *transformations* to be actually comparable. The transformations that are applicable to target cubes and benchmarks can be simple (like the above mentioned one, where the measures are computed via simple per-cell arithmetic operations), or more complex ones (like ranking or z-scoring) which require a holistic scan of the entire cube and cannot produce the new value on a per-cell basis.

We forego the formalities of the computation of the derived measures (to be discussed in Section 4.2) and simply mention that we assume a functional-style composition of the invocation of functions from our library of functions in a nestable way. For example, the min-max normalization of the difference between storeSales and target value 1000 is computed as

minMaxNorm(difference(storeSales,1000))

Listing 2 shows the implementation of these two functions in Python using Pandas DataFrames.

3.3 Labeling

The goal of this step is to associate each cell of the target cube with a label, taken from a predefined set, to express an evaluation of that cell with reference to the benchmark. Clearly, ordinal labeling will frequently be the case, however, for the sake of generality *we assume labels to be nominal*, i.e., categorical. Given a finite set of distinct values L, a labeling function has the form $\lambda : \mathbb{R} \rightarrow L$. Each value resulting from the comparison of a target cube cell with the corresponding benchmark cell is fed to the labeling function, and assigned the appropriate label.

There are some properties of interest for a labeling function:

- The labeling of comparison values is generic enough to also incorporate the labeling based on the actual value of the cell, without the usage of any benchmark and comparison. One simply needs to assign a fixed benchmark of zeros for all cells and a simple arithmetic difference as the comparison function.
- A labeling function should partition the values of the comparison into equivalence classes, i.e., there must be a *complete* mapping of the values of the domain of the comparison to a set of *non-overlapping*, disjoint labels. Thus, every cell of the result is assigned to exactly one label.
- The labeling function does not necessarily have to be predefined before the query. Assuming, for example, that a Likert-like scale based on the absolute difference value is to be adopted, the labeling function is produced after the results are obtained and split into a fixed number of groups (say 5).

With reference to the last point, in the sequel we introduce and explain two cases of labeling functions.

3.3.1 Labeling based on explicit ranges. In this case, we label each cell based on the result of the comparison between the

```
def 5stars(a):
    return pd.cut(a, [-1, -0.6, -0.2, 0.2, 0.6, 1.0],
    include_lowest = True,
    labels=["*", "**", "***", "****", "*****"])
```

Listing 3: Implementation of the 5stars function

measure values of (a) the target and (b) the benchmark cube, using a set of explicitly-specified rules. This is the case, e.g., where the organization has predetermined goals to achieve (expressed via the benchmark), and the (positive or negative) deviation from these goals characterizes the extent of success or failure.

Example 3.3. Let a query be given that computes the total store sales by customer gender, returning a target cube C with two cells, say $C = \{\langle \text{male}, 4400\rangle, \langle \text{female}, 6900\rangle\}$. Assume that we have specified an external benchmark with two cells also, $B = \{\langle \text{male}, 5400\rangle, \langle \text{female}, 6400\rangle\}$. Finally, assume that we specify a range-based labeling function called 5stars to be applied over the min-max normalized difference x of the target cube and the benchmark:

$$\lambda_{\text{5stars}}(x) = \begin{cases} *, \text{ if } -1 \leq x \leq -0.6 \\ **, \text{ if } -0.6 < x \leq -0.2 \\ ***, \text{ if } -0.2 < x \leq 0.2 \\ ****, \text{ if } 0.2 < x \leq 0.6 \\ *****, \text{ if } 0.6 < x \leq 1 \end{cases}$$

Then, the two cells are labeled as '*' and '*****', respectively. Listing 3 shows the implementation of the 5stars function in Python using Pandas DataFrames; the cut function of Pandas bins values into discrete intervals. □

3.3.2 Labeling based on the overall value distribution. Explicitly providing rules and ranges for the labels has the benefit that the decision on which label to give to a cell of the target cube is *local*, i.e., it depends only on the value of the cell's measure, the benchmark's measure, and the result of their comparison. However, the labeling function can also be based on a *holistic* assessment of the overall distribution of the values of the comparison function. In this case, the labeling function first groups the cells of the target cube based on the result of their comparison with their respective benchmark cell, and then gives a label to each group. The simplest possibility would be to split the comparison value into quartiles or, more generally, into k groups, and label each group as 'top-1', 'top-2', ..., 'top-k'. This involves simply the ranking of the values and the splitting of the ordered set of cells into k groups. Assuming a fixed set of k labels, the label is then determined by the position of a cell in the ranking.

Overall, labels can be assigned either by fixing the number of labels to a constant number and constructing equi-depth or equi-width histograms, or by allowing the system to come up with the optimal number of clusters and assign cells accordingly. More simplistic schemes (e.g., rounding the z-score of the comparison values) can also be devised. Overall, the idea of these labeling schemes is to avoid predefining ranges, and allowing labels to adapt to the distribution of the comparison values.

4 SYNTAX & SEMANTICS OF THE ASSESS OPERATOR

In this section, we formally define the syntax and semantics of the assess operator. We begin by introducing in Section 4.1 a user-friendly SQL-like syntax, to facilitate end users in posing

assessment queries with both expressive power and ease. Then, we move on to define the semantics of the assess operator in Section 4.3. To support this task, in Section 4.2 we preliminarily define a set of logical operators.

4.1 Syntax

The general syntax for writing a statement based on the assess operator includes three parts: one (consisting of the with, assess, by, and for clauses) that specifies the target cube; one (consisting of the against clause) that specifies the benchmark; one (consisting of the using and labels clauses) that specifies the assessment method. Importantly, as we will explain in Section 4.3, the benchmark specification drives the mapping of the assess syntax to the logical operators defined in Section 4.2.

with C_0 [for P] by G
assess|assess* m [against $< benchmark >$]
[using $< function >$] labels λ

where C_0 is a detailed cube (with schema $C = (H, M)$), m is a measure of C_0, P is a set of conjunctive selection predicates each over one level of H, G is a group-by set of C, $< benchmark >$ is the benchmark specification, $< function >$ specifies what will be compared and how, and λ is a labeling function (optional parts of the syntax are in brackets). While in assess only the cells of the target cube that have a match in the benchmark are returned, in the assess* variant all the cells of the target cube are returned, possibly completed with null labels.

The target cube, C, is defined by aggregating C_0 on G and selecting the cells that meet the conjunctive predicates in P.

As to the benchmark, its specification can take different forms:

- For constant benchmarks, the against clause has the form

  against v

  where v is a value compatible with m. The benchmark B is characterized by $G_B = G_C$, $P_B = P_C$. B has a measure m_{const} which takes value v in all cells. A particular case is when the user wants to *directly* assess the measure value without using any specific value. In this case the against clause is omitted; as mentioned in Section 3.3, this practically corresponds to adopting a dummy benchmark where all cells are zeros.
- For external benchmarks, the against clause takes the form

  against $B.m_b$

  where B is a cube and m_b is one of its measures. Note that C and B are joinable only if they have the same group-by set.
- In a sibling benchmark, the for clause must include a predicate which slices the target cube on member u of level $l_s \in G_C$. In this case, m is assessed against a benchmark related to a different member of l_s, say u_{sib}:

  with $< cube >$ for $p_1, \ldots, p_k, l_s = u$ by G
  assess m against $l_s = u_{sib}$
  using $< function >$ labels λ

  Here the benchmark is characterized by $G_B = G_C$ and $P_B = P_C \setminus \{p_s\} \cup \{(l_s = u_{sib})\}$. In practice, the slicing on u is replaced by one on u_{sib}.
- In a past benchmark the syntax takes the form

  with $< cube >$ for $p_1, \ldots, p_k, l_t = u$ by G
  assess m against past k
  using $< function >$ labels λ

  where l_t is a temporal level, $l_t \in G$, and k is an integer. Here the benchmark is isomorphic to C, except that the values of m are those predicted based on a time series of length v.

Finally, as to the assessment method, its specification is based on the using and labels clauses.

- The using clause specifies a (nested) function that describes how the comparison is made, including possible transformations to be made on measures (e.g., the computation of a derived measure). Here, a keyword benchmark is used to distinguish, when necessary, the cells of the target cube from the corresponding ones in the benchmark.
- The labels clause specifies a labeling function, either based on explicit ranges or on the overall value distribution, to be applied to the result of the computation specified by the using clause. A range-based labeling function can be either predeclared by the user and given a name (e.g., 5star in Example 3.3) or declared inline within the statement by listing its set of ranges with the corresponding label; the user is in charge of ensuring that the set of ranges is complete and non-overlapping. A set of library labeling function based on the value distribution (e.g., quartiles) is also made available to users.

In all cases above, the result returned to the user includes, for each cell, (i) its coordinate, (ii) the value of m for that coordinate, (iii) the value of the benchmark measure, (iv) the value resulting from the comparison, and (v) the corresponding label. The benchmark measure is m_{const} for constant benchmarks, m for sibling and past benchmarks, and m_b for external benchmarks.

Example 4.1. The first example gives an absolute assessment of the total monthly sales in terms of quartiles:

```
with SALES by month
assess storeSales labels quartiles
```

Similarly, sales can be assessed against a goal value, say 1000, via a 5 star scale in the [0..1] range by first normalizing the difference and then using the range-based labeling function specified in Example 3.3:

```
with SALES by month
assess storeSales against 1000
using minMaxNorm(difference(storeSales,1000))
labels 5star
```

The following statement uses a sibling benchmark; for each product of type fresh fruit, the total quantity sold in Italy is assessed against the one in France. For each product, assessment is based on the ratio between (i) the difference in quantities sold in Italy and France, and (ii) the total sales of fresh fruit in Italy; this ratio

is computed using library function *percOfTotal*.

```
with SALES
for type = 'Fresh Fruit', country ='Italy'
by product, country
assess quantity against country = 'France'
using percOfTotal(difference(quantity, benchmark.quantity))
labels {[-inf, -0.2): bad, [-0.2,0.2]: ok, (0.2, inf]: good},
```

Finally, in the next statement we use a past benchmark; specifically, we assess the sales of a specific store in July 1997 against the past four months:

```
with SALES
for month = '1997-07', store = 'SmartMart'
by month, store
assess storeSales against past 4
using ratio(storeSales, benchmark.storeSales)
labels {[0, 0.9): worse, [0.9, 1.1]: fine, (1.1,inf): better}
```

□

4.2 Logical operators

This section introduces the logical aspects behind the different steps of the evaluation of an assess statement, formulated as logical operators. Note that our aim is not to propose a logical language for manipulating cubes (such languages exist, see e.g. [2]) but to describe specific cube manipulations required to logically optimize assess statements. In particular, we do not detail the classical (roll-up, etc.) cube manipulations.

We recall from Section 2 that a cube is defined as a partial function that maps coordinates into tuples of measures. For a cube C and a coordinate γ such that $C(\gamma) = t$, we denote with $c = \langle \gamma, t \rangle$ the cell defined by $C(\gamma)$ and we abusively note $c \in C$. We define operators that respect the closure property, in the sense that they operate on cubes and specify cubes.

Get. The first basic operator consists of obtaining the result of a cube query. Given a cube C over a schema $\mathcal{C} = (H, M)$, a set of selection predicates P and a group-by set G of $\mathcal{C}$, the get operator corresponds to the cube query $q = (C, G, P, M)$, is denoted by $[q]$, and defines the derived cube being the result of q. Note that $[(C, G_0, \emptyset, M)]$ is simply noted $[C]$ in what follows. Besides, the derived cube returned by get can be renamed using the notation $[(C, G, P, M)] \rightarrow name$.

Join $\boxtimes$. The join operation is essential for putting together the target cube (C_1) and the benchmark (C_2). In OLAP terms this is a drill-across operation, or join applied to cubes.

Let C_1 and C_2 be two joinable cubes over schemas $\mathcal{C}_1$ and $\mathcal{C}_2$. As already stated, we assume for simplicity that the two cubes share the same hierarchies, so that $\mathcal{C}_1 = (H, M_1)$ and $\mathcal{C}_2 = (H, M_2)$.

$$C_1 \boxtimes C_2 = \{\langle \gamma, (t, t') \rangle | \langle \gamma, t \rangle \in C_1, \langle \gamma, t' \rangle \in C_2\}$$

The schema of the resulting cube is $(H, (M_1, M_2))$.

We also define a version of join where we allow partial joining in the sense that join is made on a subset of the levels of H. Formally:

$$C_1 \boxtimes_{l_1,\ldots,l_m} C_2 = \{\langle \gamma, (t, t^1, \ldots, t^p) \rangle | \\ \langle \gamma, t \rangle \in C_1, \langle \gamma^j, t^j \rangle \in C_2, \gamma_{|_{l_1,\ldots,l_m}} = \gamma^j_{|_{l_1,\ldots,l_m}}, j \in [1, \ldots, p]\}$$

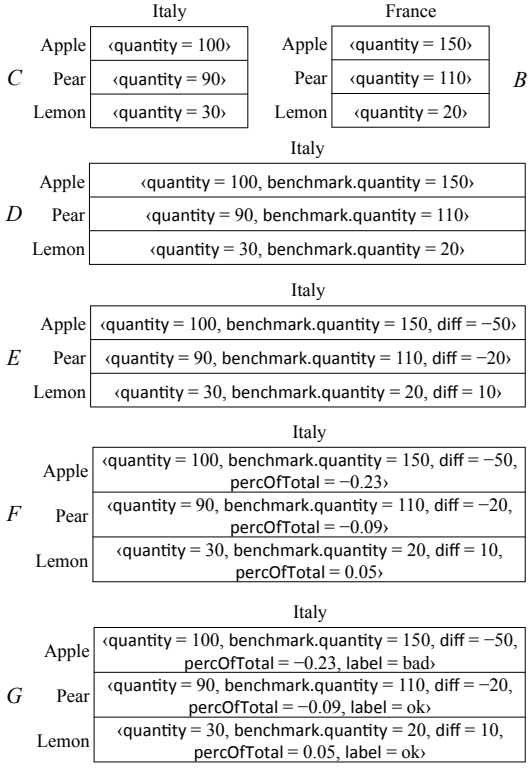


Figure 1: Derived cubes resulting from the application of logical operators for the sibling intention in Example 4.5

Note that, differently from the (natural) join defined above, this partial join is not commutative.

Finally, the assess* syntactical variant (that also returns the non-matching cells of the target cube) uses a left-outer join $C_1 \ ^* \boxtimes \ C_2$ where non-matching cells are completed with null values.

Example 4.2. Figure 1 shows the results, C and B, of the following get operations:

```
C =[(SALES, ⟨product, country⟩,
    {type = 'Fresh Fruit', country = 'Italy'},
    ⟨quantity⟩)]
B =[(SALES, ⟨product, country⟩,
    {type = 'Fresh Fruit', country = 'France'},
    ⟨quantity⟩)] → benchmark
```

(cube B is given alias benchmark) and the result of their partial join, $D = C \boxtimes_{\text{product}} B$. □

Cell-Transform $\boxminus$. This operator specifies a cell-at-a-time operation that takes a cube and a function, and outputs a cube where a new measure is added, containing the value of the function applied over the measure(s). Let $\mathcal{C} = (H, M)$ be the schema of cube C with group-by set G, and $\overline{M}$ be a subtuple of M. Let f be a function defined on a tuple of parameters compatible with $\overline{M}$; then the cell-transformation operation operated by f returns a cube defined by:

$$\boxminus_{f \rightarrow name, \overline{M}}(C) = \{\langle \gamma, (t, \langle f(\overline{M}) \rangle) \rangle \mid \langle \gamma, t \rangle \in C\}$$

The schema of the resulting cube is $(H, (M, \langle name \rangle))$, where *name* is the derived measure returned by f.

C'

	Italy	France
Apple	‹quantity = 100›	‹quantity = 150›
Pear	‹quantity = 90›	‹quantity = 110›
Lemon	‹quantity = 30›	‹quantity = 20›

D'

	Italy
Apple	‹quantity = 100, qtyFrance = 150›
Pear	‹quantity = 90, qtyFrance = 110›
Lemon	‹quantity = 30, qtyFrance = 20›

Figure 2: Example of application of the pivot operator

H-Transform $\boxdot$. This operator considers holistic (H) transformations, in the sense that computing a new measure value for each cell of cube C requires to know *all* the cells of C.

Let again $\overline{M}$ be a subtuple of M as above. In this case, function f operates on a tuple of parameters compatible with $\overline{M}$ *and* on a set of tuples. The H-transformation of C operated by f returns a cube defined by:

$$\boxdot_{f \to name,\overline{M}}(C) = \{\langle \gamma, (t, \langle f(\overline{M}, C)\rangle)\rangle \mid \langle \gamma, t\rangle \in C\}$$

The schema of the resulting cube is $(H, (M, \langle name\rangle))$, where *name* is the extra measure returned by f.

Example 4.3. The following cell-transformation extends cube D with a derived measure storing their difference:

$$E = \boxminus_{difference \to \text{diff},\langle \text{quantity,benchmark.quantity}\rangle}(D)$$

Then, cube F is obtained from E by applying an H-transformation as follows:

$$F = \boxdot_{percOfTotal \to \text{percOfTotal},\langle \text{diff,quantity}\rangle}(E)$$

where holistic function $percOfTotal$ operates on a tuple of two parameters a and b and computes, for each cell, the ratio between a and the sum of b over all cells. □

Pivot $\boxplus$. This operator takes a cube including a set of k slices of some level l (a cube *slice* is the set of cells corresponding to one single member of a level), among which only one slice for a given member $u_k \in Dom(l)$ is returned. Each coordinate γ of this slice in the returned cube is associated with its initial tuple of measures t, concatenated with all the p measures $\overline{M} = \langle m^1, \ldots, m^p\rangle$ of all its $k-1$ neighbor coordinates γ' in the initial set of k slices. The new measures are renamed $name^1, \ldots, name^p$. Formally, given cube C with schema $C = (H, M)$, let $u_1, \ldots, u_k \in Dom(l)$ be the members of l on which the slices are defined. Let u_k be the reference slice for pivoting. Then

$$\boxplus_{\langle m^1 \to name^1, \ldots, m^p \to name^p\rangle, l, u_k}(C) = \{\langle \gamma, (t, \langle v_1^1, \ldots, v_{k-1}^1, \ldots, v_1^p, \ldots, v_{k-1}^p\rangle)\rangle \mid \langle \gamma, t\rangle \in C, \gamma_{|_l} = u_k, \langle \gamma', t'\rangle \in C, \gamma'_{|_l} = u_i, \gamma_{|_{G\setminus l}} = \gamma'_{|_{G\setminus l}}, t'_{|_{m^j}} = v_i^j, i \in [1, k-1], j \in [1, p]\}$$

where the t's are tuples of measure values. The schema of the resulting cube is $(H, (M, name^1, \ldots, name^p))$ where in turn $name = \langle m_1, \ldots, m_{k-1}\rangle$.

Example 4.4. Figure 2 shows the result C' of the following get operator:

$C' =$[(SALES, ⟨product, country⟩,
{type = 'Fresh Fruit', country ∈ {'Italy', 'France'}},
⟨quantity⟩)]

Cube C' includes two slices for country. By applying the following pivot operator:

$$D' = \boxplus_{\langle \text{quantity}\rangle \to \text{qtyFrance,country,'Italy'}}(C')$$

a cube D' is obtained that includes only the reference slice ('Italy'), with an extra measure qtyFrance. □

4.3 Semantics

Assume the expression of the assess operator as defined in Section 4.1:

with C_0 [for P] by G
assess m [against < *benchmark* >]
[using < $function$ >] labels λ

In terms of the logical operators introduced in Section 4.2, let

(1) $\boxdot_{\Delta,}(\cdot)$ be the composition of the comparison/transformation functions denoted by the using clause.
(2) $\boxdot_{\lambda,}(\cdot)$ be the transformation that applies the labeling function denoted by the labels clause.

Without loss of generalization, we assume that the functions that are used for the comparison and the labeling are holistic. Clearly, the application of cell-based functions is also possible (and most welcome for efficiency and optimization purposes).

The semantics of an assess statement is defined as

$$\boxdot_{\lambda \to m^\lambda, m^\Delta}(\boxdot_{\Delta \to m^\Delta, \overline{M}}(C))$$

where the definition of cube C depends on the type of benchmark used, which in turn is determined by the form taken by the against clause as explained in Section 4.1:

- Constant benchmark: $C = [(C_0, G, P, M)]$.
- External benchmark B: $C = [(C_0, G, P, M)] \boxtimes [B]$
- Sibling benchmark:

$$C = [(C_0, G, P, M)] \boxtimes_{G\setminus l_s} [(C_0, G, P_B, M)] \to \text{benchmark}$$

where $P_B = P \setminus \{(l_s = u)\} \cup \{(l_s = u_{sib})\}$.

- Past benchmark:

$$C = [(C_0, G, P, M)] \boxtimes_{G\setminus l_t} (\boxminus_{regression \to M', M}(\boxplus_{M \to M', l_t, u}([(C_0, G, P_B, M)] \to \text{benchmark})))$$

where $P_B = P \setminus \{(l_t = u)\} \cup \{(l_t \in \{u_1, \ldots, u_k\})$, members $u_1, \ldots, u_k$ are predecessors of u for level l_t, and $regression$ is a time series prediction function.

Note that, in the assess* variant, the inner join is replaced by a left-outer join. In all cases, the resulting cube has schema $(H, \langle m, m^B, m^\Delta, m^\lambda\rangle)$ The benchmark measure m^B is m_{const} for constant benchmarks, m for sibling and past benchmarks, and m_b for external benchmarks.

Example 4.5. Consider again some of the statements of Example 4.1. The first one relies on a constant benchmark:

with SALES by month
assess storeSales labels quartiles,

and corresponds to the logical expression:

$$\boxdot_{quartiles,\langle \text{storeSales}\rangle}([(\text{SALES}, \langle \text{month}\rangle, \emptyset, \langle \text{storeSales}\rangle)])$$

The one based on a sibling benchmark,

```
with SALES
for type = 'Fresh Fruit', country ='Italy'
by product, country
assess quantity against country = 'France'
using percOfTotal(difference(quantity, benchmark.quantity))
labels {[-inf, -0.2): bad, [-0.2,0.2]: ok, (0.2, inf]: good},
```

corresponds to the following plan (see Figure 1):

(1) get the target cube:

$$C = [(\text{SALES}, \langle \text{product}, \text{country} \rangle, \{\text{type} = \text{'Fresh Fruit'}, \text{country} = \text{'Italy'}\}, \langle \text{quantity} \rangle)]$$

(2) get the benchmark:

$$B = [(\text{SALES}, \langle \text{product}, \text{country} \rangle, \{\text{type} = \text{'Fresh Fruit'}, \text{country} = \text{'France'}\}, \langle \text{quantity} \rangle)] \rightarrow \text{benchmark}$$

(3) (partially) join C and B:

$$D = C \boxtimes_{\text{product}} B$$

(4) transform D:

$$E = \boxminus_{difference \rightarrow \text{diff}, \langle \text{quantity,benchmark.quantity} \rangle}(D)$$

(5) transform E:

$$F = \boxdot_{percOfTotal \rightarrow \text{percOfTotal}, \langle \text{diff,quantity} \rangle}(E)$$

(6) transform F:

$$G = \boxminus_{range(\{[-\text{inf},-0.2):\text{bad}, [-0.2,0.2]:\text{ok}, (0.2,\text{inf}]:\text{good}\}), \langle \text{percOfTotal} \rangle}(F)$$

The last one uses a past benchmark:

```
with SALES
for month = '1997-07', store = 'SmartMart'
by month, store
assess storeSales against past 4
using ratio(storeSales, benchmark.storeSales)
labels {[0, 0.9): worse, [0.9, 1.1]: fine, (1.1,inf): better}
```

and corresponds to the following plan:

(1) get the target cube:

$$C = [(\text{SALES}, \langle \text{month}, \text{store} \rangle, \{\text{month} = \text{'1997-07'}, \text{store} = \text{'SmartMart'}\}, \langle \text{storeSales} \rangle)]$$

(2) get the data for the benchmark:

$$B = [(\text{SALES}, \langle \text{month}, \text{store} \rangle, \{\text{month} \in [\text{'1997-03'}; \text{'1997-06'}], \text{store} = \text{'SmartMart'}\}, \langle \text{storeSales} \rangle)] \rightarrow \text{benchmark}$$

(3) pivot B:

$$D = \boxplus_{\langle \text{storeSales} \rangle \rightarrow \text{past,month,'1997-06'}} B$$

(4) transform D:

$$E = \boxminus_{regression \rightarrow \langle \text{storeSales} \rangle, \text{past}} D$$

(5) (partially) join C and E:

$$F = C \boxtimes_{\text{store}} E$$

(6) transform F:

$$G = \boxminus_{ratio \rightarrow \text{r}, \langle \text{storeSales,benchmark.storeSales} \rangle} F$$

(7) transform G:

$$\boxminus_{range(\{[0,0.9):\text{worse}, [0.9,1.1]:\text{fine}, (1.1,\text{inf}):\text{better}\}), \langle \text{r} \rangle} G$$

5 OPTIMIZING ASSESS STATEMENTS

This section illustrates how the logical operators introduced above allow to optimize the evaluation strategies of assess in a rule-based fashion. We start by giving basic algebraic properties of the operators, and then present optimization schemes exploiting these properties.

5.1 Basic properties

Commutativity of transform (P_1). An important feature of the transform operators is that they preserve the set of coordinates of the cube they are applied to, monotonically adding new measures to it. In other words, the operators commute when one does not need the result of the other. Formally,

$$\boxminus_{f \rightarrow n_f, M'}(\boxminus_{g \rightarrow n_g, M}(C)) = \boxminus_{g \rightarrow n_g, M}(\boxminus_{f \rightarrow n_f, M'}(C))$$

if $n_g \notin M'$ and $n_f \notin M$. The same property holds for $\boxdot$, and for combinations of $\boxdot$ and $\boxminus$.

Pushing join through transformation (P_2). A join can be pushed before a cell-transformation, if the transformation is applied to the measures of only one of the joined cubes, by applying the transformation directly over that cube and removing the pivot operation needed to guarantee the two cubes are joinable. Formally,

$$(C, G, P, M) \boxtimes_{G \setminus \{l\}} (\boxminus_{f \rightarrow n_f, M_2} \boxplus_{M_1 \rightarrow M_2, l, u} (C, G, P', M_1)) = \boxminus_{f \rightarrow n_f, M_1}((C, G, P, M) \boxtimes_{G \setminus \{l\}} (C, G, P', M_1))$$

where $P' = P \setminus \{(l_s = u)\} \cup \{(l_s \in \{u_1, \ldots, l_n\})$.

Replacing join with pivot (P_3). Joining different slices of the same cube can be done either by getting each slice individually and partially joining them, or by getting the slices together and pivoting all but one of them. Formally,

$$[(C, G, P, M)] \boxtimes_{G \setminus \{l\}} [(C, G, P', M)] = \boxplus_{M \rightarrow M', l, u} [(C, G, P_{all}, M)]$$

where M' is a tuple of measure names not in M, $P' = P \setminus \{(l = u)\} \cup \{(l \in \{u_1, \ldots, u_n\})$ and $P_{all} = P \setminus \{(l = u)\} \cup \{(l \in \{u, u_1, \ldots, u_n\})\}$.

5.2 Optimization strategies

In a classical interactive cube analysis, a user expresses high-level manipulations through front-end applications over DBMSs. We assume the same context, where cubes are accessed through cube queries (our logical get operation), over an already properly optimized DBMS. In this setting, we work under the following hypotheses: (i) the get, join, and pivot logical operations can be executed via SQL queries; (ii) the results of these SQL queries fit in main memory; (iii) all transformations are seen as black-box functions, thus they are not pushed to SQL. The optimization opportunities of assess statements are then related to which logical operators are pushed to SQL.

Following the above assumptions, for an assess statement we consider three possible plans, based on different execution strategies, as described in the following subsections.

```
select t1.country, t1.product,
       t1.quantity, t2.quantity as bc_quantity
from
  (select country, product, sum(quantity) as quantity
   from sales s
     join customer c on c.ckey = s.ckey
     join product p on p.pkey = s.pkey
   where type = 'Fresh Fruit' and country = 'Italy'
   group by country, product) t1,
  (select country, product, sum(quantity) as quantity
   from sales s
     join customer c on c.ckey = s.ckey
     join product p on p.pkey = s.pkey
   where type = 'Fresh Fruit' and country = 'France'
   group by country, product) t2
where t1.product = t2.product
```

Listing 4: Getting the pivoted cube of the sibling intention following JOP

5.2.1 Naive Plan. A *Naive Plan* (NP) faithfully reproduces the sequences of operations shown in Section 4.3; only the get operations are pushed to SQL and all other operations are executed in memory. NP is feasible for all benchmark types.

Example 5.1. Consider the sibling statement of Example 4.5. Its NP consists in translating individually each get operation into an SQL call, to retrieve the target and benchmark cubes. Specifically, the first get operation is translated in the SQL query of Listing 1; the second get operation consists of the same SQL code where the selection is made on 'France' instead of 'Italy'. All other subsequent operations of that statement, i.e., the partial join and the transformations, are done in memory.

5.2.2 Join-Optimized Plan. In a *Join-Optimized Plan* (JOP), also the join is pushed to SQL to take advantage of the DBMS optimizer. This requires that the plan starts with the subexpression $C \boxtimes B$, where C and B are two get operations, so that all three operations can be pushed to SQL. JOP is not feasible for constant benchmarks, since there is no join to be done; for the other benchmark types, it may require property P_2 to be applied to NP to postpone cell-transformations after the join.

Example 5.2. Consider the sibling statement of Example 4.5, and the subexpression of step (3): $D = C \boxtimes_{\text{product}} B$. This subexpression is translated to the SQL query of Listing 4, with one inner subquery for each get operation C and B, and an outer query for joining them. □

Example 5.3. As mentioned above, property P_2 can be used to put an assess statement in a form that allows pushing the join to SQL. Consider for instance the five first steps of the past statement of Example 4.5. Applying property P_2 turns these steps into the plan:

(1) get the target cube:

$$C = [(\text{SALES}, \langle \text{month}, \text{store} \rangle, \{\text{month} = \text{'1997-07'}, \text{store} = \text{'SmartMart'}\}, \langle \text{storeSales} \rangle)]$$

(2) get the data for the benchmark:

$$B = [(\text{SALES}, \langle \text{month}, \text{store} \rangle, \{\text{month} \in [\text{'1997-03'}; \text{'1997-06'}], \text{store} = \text{'SmartMart'}\}, \langle \text{storeSales} \rangle)] \rightarrow \text{benchmark}$$

(3) (partially) join C and B:

$$D = C \boxtimes_{\text{store}} B$$

(4) transform D:

$$E = \boxminus_{regression \rightarrow \langle \text{storeSales} \rangle, \text{benchmark.storeSales}} D$$

The subexpression $D = C \boxtimes_{\text{store}} B$ can be then pushed to SQL. □

5.2.3 Pivot-Optimized Plan. The goal of a *Pivot-Optimized Plan* (POP) is to let the DBMS compute pivot operations. To this end, whenever the plan starts with the subexpression $C \boxtimes B$, where C and B are get operations on the same cube, the join operation is replaced with the pivot operation using property P_3, resulting in a pivot operation for aligning the target and benchmark slices. Both operations (get and pivot) are then pushed to SQL. POP is feasible only for sibling and past intentions, which get multiple slices from a single cube.

Example 5.4. Consider the sibling statement of Example 4.5. Using property P_3 allows to rewrite the plan to (see also Figures 1 and 2):

(1) get the (target+benchmark) cube:

$$C' = [(\text{SALES}, \langle \text{product}, \text{country} \rangle, \{\text{type} = \text{'Fresh Fruit'}, \text{country} \in \{\text{'Italy'}, \text{'France'}\}, \langle \text{quantity} \rangle)]$$

(2) pivot C':

$$E = \boxplus_{\langle \text{quantity} \rightarrow \text{qtyFrance} \rangle, \text{country}, \text{'Italy'}} (C')$$

(3) transform D':

$$E' = \boxminus_{difference \rightarrow \text{diff}, \langle \text{quantity}, \text{qtyFrance} \rangle} (D')$$

(4) transform E':

$$F' = \boxdot_{percOfTotal \rightarrow \text{percOfTotal}, \langle \text{diff}, \text{quantity} \rangle} (E')$$

(5) transform F':

$$G' = \boxminus_{range(\{[-\text{inf}, -0.2):\text{bad}, [-0.2, 0.2]:\text{ok}, (0.2, \text{inf}]:\text{good}\}), \langle \text{percOfTotal} \rangle} (F')$$

Listing 5 shows the resulting SQL query. Likewise, the past statement, in the form given in Example 5.3, can be rewritten with P_3 as:

(1) get (target+benchmark) cube:

$$D = [(\text{SALES}, \langle \text{month}, \text{store} \rangle, \{\text{month} \in [\text{1997-03}; \text{1997-07}], \text{store} = \text{'SmartMart'}\}, \langle \text{storeSales} \rangle)]$$

(2) pivot D:

$$E = \boxplus_{\langle \text{storeSales} \rangle \rightarrow \text{past}, \text{month}, \text{'1997-07'}} (D)$$

(3) transform E:

$$F = \boxminus_{regression \rightarrow \text{benchmark.storeSales}, \langle \text{past} \rangle} (E)$$

(4) transform F:

$$G = \boxminus_{ratio \rightarrow \text{r}, \langle \text{storeSales}, \text{benchmark.storeSales} \rangle} (F)$$

(5) transform G:

$$\boxminus_{range(\{[0,0.9):\text{worse}, [0.9,1.1]:\text{fine}, (1.1, \text{inf}):\text{better}\}), \langle \text{r} \rangle} (G)$$

Under this form, the first two steps of the plan can be transformed into SQL calls. □

```sql
select 'Italy' as country, product,
  quantity, bc_quantity
from
  (select country, product, sum(quantity) as quantity
   from sales s
     join customer c on c.ckey = s.ckey
     join product p on p.pkey = s.pkey
   where type = 'Fresh Fruit'
         and country in ('Italy', 'France')
   group by country, product)
pivot (
  sum(quantity) for country
    in ('Italy' as quantity, 'France' as bc_quantity)
)
where quantity is not null and bc_quantity is not null
```

Listing 5: Getting the pivoted cube of the sibling intention following POP

Table 1: Formulation effort for different intentions

	Constant	External	Sibling	Past
SQL:	481	989	1169	1954
Python:	7006	6193	6309	7049
Total:	7487	7182	7478	9003
assess:	143	260	270	254

6 EXPERIMENTS

To test our approach, we implemented the assess operator relying on the simple multidimensional engine described in [6], which uses multidimensional metadata to rewrite OLAP queries on a star schema stored in Oracle 11g DBMS. Post-processing of the results (e.g., to apply transformations) is then done via off-the-shelf Python Scikit-learn over Pandas DataFrames. All tests were run on an Intel(R) Core(TM)i7-6700 CPU@3.40GHz CPU with 8GB RAM.

The prototype was tested against the Star Schema Benchmark (SSB) cube, described by four hierarchies; please refer to [14] for the logical schema of the SSB dataset. As commonly done in OLAP settings, primary and foreign keys were indexed using B-Trees, and materialized views were created to improve performances. The experiments are focused on four assess statements of different types, henceforth referred to as Constant, External, Sibling, and Past, respectively.

6.1 Formulation effort

The first goal of our experiments is to evaluate the saving in user's effort when writing an assess statement over the one necessary to obtain the same result using plain SQL and Python. To this end we adopt the simple metric proposed in [11], where the ASCII character length is used as as a proxy for the effort it takes to craft a query. The results are shown in Table 1. For SQL and Python we considered the code generated by our prototype when following the less complex plan. Nevertheless, as expected, the total formulation effort using SQL+Python is, for each intention type, more than one order of magnitude larger than using assess statements.

6.2 Efficiency and scalability

Our second experimental goal is to evaluate the efficiency of our approach in executing (i) different types of intentions, (ii) with different execution plans, and (iii) on cubes with different cardinalities. To achieve (iii) we generated three detailed SSB cubes, namely SSB_1, SSB_{10}, and SSB_{100}, with different scale factors resulting in the following cardinalities:

$$|SSB_1| = 6 \cdot 10^6$$
$$|SSB_{10}| = 6 \cdot 10^7$$
$$|SSB_{100}| = 6 \cdot 10^8$$

Note that the cardinality of each cube is equal to the number of tuples in the corresponding fact table. Since the by and for clauses of each assess statement are not changed, scaling up the cardinality of the detailed cube implies that also the cardinality of the target cube scales up as shown in Table 2. To reduce the impact of caching, each assess statement was executed five times on each detailed cube, and the execution times were averaged.

Table 2: Target cube cardinalities for each intention type applied to each detailed cube

	SSB_1	SSB_{10}	SSB_{100}
Constant	$1.2 \cdot 10^5$	$1.2 \cdot 10^6$	$1.2 \cdot 10^7$
External	$2.4 \cdot 10^4$	$2.5 \cdot 10^5$	$2.5 \cdot 10^6$
Sibling	$2.4 \cdot 10^4$	$2.5 \cdot 10^5$	$2.5 \cdot 10^6$
Past	$1.5 \cdot 10^3$	$1.6 \cdot 10^4$	$1.6 \cdot 10^5$

Table 3: Minimum execution times (in seconds) for different intentions (in parentheses, the corresponding execution times for NP)

	SSB_1	SSB_{10}	SSB_{100}
Constant	0.60 (0.60)	6.77 (6.77)	45.14 (45.14)
External	0.27 (0.31)	2.38 (2.60)	32.86 (35.60)
Sibling	0.32 (0.42)	3.69 (4.97)	49.61 (99.93)
Past	1.20 (3.21)	11.72 (30.93)	118.25 (321.11)

Figure 3 shows, on a logarithmic scale, the times in seconds for executing the Constant, External, Sibling, and Past intentions using the NP, JOP, and POP plans, for increasing cube cardinalities. As to Constant, assessing a target cube of $1.2 \cdot 10^7$ tuples (derived by querying SSB_{100}) takes about 45 seconds, mostly employed to get the data from the DBMS. Note that, since this assessment does not require the retrieval of a benchmark cube, only NP is feasible. As to External, the only possible plans are NP and JOP (POP is not feasible here), with JOP providing the best performance. As to Sibling and Past, POP performs the best, taking 50 seconds and 118 seconds, respectively. Being based on the pivot operator, POP gets in both cases the target cube and the benchmark at once by retrieving the slices required together. In other words, POP avoids the join between the target cube and the benchmark, a time-consuming operation for NP and JOP. Overall, NP has the worst performance, since (i) it requires to separately get both cubes and join them into main memory, and (ii) it may load into main memory unnecessary data (i.e., the tuples that will not match in the join). Overall, we can conclude that (i) JOP, when applicable, outperforms NP, and (ii) POP, when applicable, outperforms JOP and NP. This is summarized in Table 3 which, for each benchmark type, compares the best performance with the one of the naive execution strategy. Remarkably, this table also clearly shows that our approach scales linearly for all the intentions.

Our last experimental goal is to understand which are the most expensive execution steps, i.e., those for which there is room for further optimizations. The overall execution time for

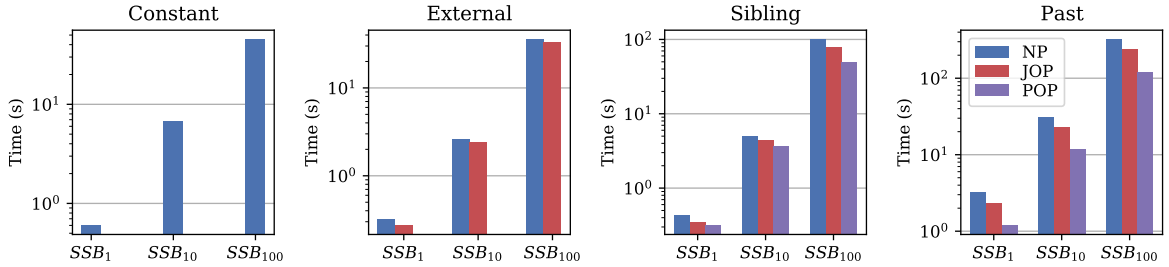


Figure 3: Execution times for increasing cardinalities of the target cube C

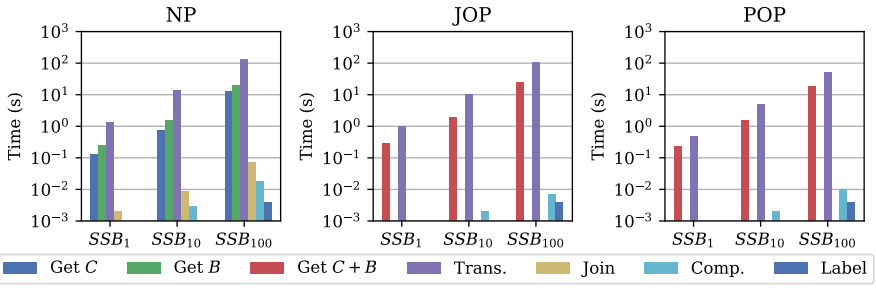


Figure 4: Breakdown of the execution time of the Past intention for increasing cardinalities of the target cube C

an intention can be broken down into the time necessary to (1) get the target cube, (2) get the benchmark, (3) transform the two cubes (e.g., to apply regression in past benchmarks), (4) join them, (5) compute the comparison/transformation, and (6) label the result. This breakdown is shown in Figure 4 for each plan and with increasing cube cardinalities. We focus on the Past intention, that is the most complex one since forecasting measure values requires to compute a regression. First of all, we observe that the execution times for comparison and labeling are in the order of milliseconds. Thus, not surprisingly, they are negligible with respect to the time necessary to get and join the cubes. For all three plans, transformation is the most time-consuming step, since linear regression has to be applied to huge numbers of tuples. As to the time for accessing data, note that:

- NP brings both the target C and benchmark B cubes into main memory to join them; the cost for this main-memory join is lower than the ones for getting the two cubes, but still not negligible. The cost for the pivot operation is counted as transformation.
- JOP pushes the join to SQL; thus, in this case the cost for the join is counted together with the one for getting $C+B$. The cost for the pivot operation is counted as transformation.
- POP replaces the join with a pivot operation and pushes it to SQL, thus, the cost of pivot here is part of the cost for getting $C+B$.

7 RELATED WORK

7.1 OLAP models and operators

OLAP comes with a large number of proposals on its foundations and operators, all of which slowly converged towards the core ideas of cubes, dimensions, dimension hierarchies, and levels as well as operators like roll-up, drill-down, slice, drill-across during the late '90s. To avoid overcrowding the discussion, we refer the interested reader to an excellent survey [16].

Over the years, several operators have been proposed to complement the fundamental ones. The *DIFF* operator [17] returns the set of tuples that most successfully describe the difference of values between two cells of a cube that are given as input. The same author also describes a method that profiles the exploration of a user and uses the Maximum Entropy principle to recommend which unvisited parts of the cube can be the most surprising in a subsequent query [18]. Finally, the *RELAX* operator allows to verify whether a pattern observed at a certain level of detail is present at a coarser level of detail too [19].

In a different line of research, prediction cubes are proposed with the characteristic property that each of the cells comes with a model that is trained to produce a predictive model with data that correspond to that cell [5]. Then, a comparison between model and actual value is also possible, assessing the model's accuracy. Also, the Shrink operator [9, 15] has been proposed to reduce the result size of a query with minimal loss of information value via the calculated fusion of data slices.

Alternative operators have also been proposed in the Cinecubes method [7, 8]. The goal of this effort is to facilitate automated reporting, given an original OLAP query as input. To achieve this purpose two operators (expressed as *acts*) are proposed, namely, (a) *put-in-context*, i.e., compare the result of the original query to query results over similar, sibling values; and (b) *give-details*, where drill-downs of the original query's groupers are performed.

Compared to the previous proposals, our work on the explicit introduction of an assess operator differs in the fundamental problem it addresses. The works of Sarawagi are mostly of explanatory rather than assessment nature. Similarly, prediction cubes are trying to assess the impact of a set of predictor attributes on a class label in the context of a data cube, via an introduced model for their relationship —again, the emphasis is on trying to explain what we see rather than trying to provide assessments and labels on the comparison of the assessment. The Shrink operator is intended to compress without losing too

much information. The Cinecubes approach introduces an automatically invoked model of assessment in its put-in-context act; this is indeed a first form of assessment, although not tunable or explicitly invoked by the user.

7.2 The Intentional Analytics Model

The IAM for OLAP was introduced in [20]. Later this proposal was significantly extended [21]. The main idea behind the intentional model for OLAP is that OLAP models need to be extended with (a) new operators, (b) altering of the definition of a query result, (c) introducing highlights to annotate the answers. To address the first requirement, the traditional roll-ups and drill-downs operators were complemented with operators that pertain to the *intention* of the user towards the data —i.e., what is the reason why the user poses the query. The original, large set of operators (including operators like verify and analyze) was later solidified and formalized into five operators, namely, describe, assess, explain, predict, and suggest [21]. The result of a query is also redefined as a combination of data and KDD *models* that are applied over the data. Also, the resulting data and models are evaluated with respect to their interestingness to produce highlights, i.e., subsets of the data that provide the most of novel information to the user. The foundations of the model can be linked to Bloom's taxonomy and Anderson and Krathwohl's refinement to it [13, 22], which organize cognitive tasks as: (a) remembering, (b) understanding, (c) applying a procedure, (d) analyzing (component interrelationships), (e) evaluating (with respect to criteria and standards), and (f) creating.

Although the IAM acts as an all-encompassing framework for defining new operators, results, and highlights for OLAP, the goal of the previous works was not go down into the details of each operator, but rather to dictate templates on what kind of algebraic operators we can introduce. The particularities of the describe operator (supporting the understanding process in Bloom's framework) were further explored [4]. The current paper extends the originally proposed assess operator (in turn, inspired by the put-in-context operator) in significantly deeper ways, as it comes with several alternatives that were not obviously expressed in the original work [21], as well as with the syntax of an SQL-like language and optimization techniques.

8 CONCLUSIONS

In this paper we have introduced the assess operator to automatically evaluate and characterize the result of a cube query in terms of labels given to the single cells based on their comparison with a benchmark. We have provided several alternatives for specifying benchmarks, comparison, and labeling schemes. Finally, we have discussed alternative plans for the execution of assess statements showing that their performance is perfectly in line with the *right time* requirement of analysis sessions.

Our future work on the assess operator will develop in different directions:

- Consider cube schemas including descriptive properties of levels (e.g., the population of a country). Introducing properties will enable users to express more complex statements, e.g., to compare *per capita* sales of different countries.
- Devise strategies for effectively completing partial assess statements, for instance, ones where the against, using or benchmark clauses are not specified by the user. Interestingly, this could require different possibilities to be tested and ranked based on their expected interest for the user.
- Enhance the expressiveness of the assess operator by considering more complex labeling functions (e.g., functions based on ranges that depend not only on comparison values of cells, but also on their coordinates) and additional types of benchmarks (for instance to let the sales of milk be assessed against those of drinks, i.e., against an ancestor of milk in the roll-up order).
- Investigate the relevant properties of our logical operators and develop a cost-based optimization strategy.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[2] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. 1997. Modeling Multidimensional Databases. In *Proceedings of ICDE*. Birmingham, UK, 232–243.

[3] Leilani Battle, Michael Stonebraker, and Remco Chang. 2013. Dynamic reduction of query result sets for interactive visualizaton. In *Proceedings of International Conference on Big Data.* Santa Clara, CA, USA, 1–8.

[4] Antoine Chédin, Matteo Francia, Patrick Marcel, Verónika Peralta, and Stefano Rizzi. 2020. The Tell-Tale Cube. In *Proceedings of ADBIS*. Lyon, France, 204–218.

[5] Bee-Chung Chen, Lei Chen, Yi Lin, and Raghu Ramakrishnan. 2005. Prediction Cubes. In *Proceedings of VLDB*. Trondheim, Norway, 982–993.

[6] Matteo Francia, Enrico Gallinucci, and Matteo Golfarelli. 2020. Towards Conversational OLAP. In *Proceedings of DOLAP*. Copenhagen, Denmark, 6–15.

[7] Dimitrios Gkesoulis and Panos Vassiliadis. 2013. CineCubes: cubes as movie stars with little effort. In *Proceedings of DOLAP*. San Francisco, CA, USA, 3–10.

[8] Dimitrios Gkesoulis, Panos Vassiliadis, and Petros Manousis. 2015. CineCubes: Aiding data workers gain insights from OLAP queries. *Inf. Syst.* 53 (2015), 60–86.

[9] Matteo Golfarelli, Simone Graziani, and Stefano Rizzi. 2014. Shrink: An OLAP operation for balancing precision and size of pivot tables. *Data Knowl. Eng.* 93 (2014), 19–41.

[10] Matteo Golfarelli, Federica Mandreoli, Wilma Penzo, Stefano Rizzi, and Elisa Turricchia. 2012. OLAP query reformulation in peer-to-peer data warehousing. *Inf. Syst.* 37, 5 (2012), 393–411.

[11] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. 2016. SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment. In *Proceedings of SIGMOD*. San Francisco, CA, USA, 281–293.

[12] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE Trans. Vis. Comput. Graph.* 18, 12 (2012), 2917–2926.

[13] David R. Krathwohl. 2002. A Revision of Bloom's Taxonomy: An Overview. *Theory Into Practice* 41, 4 (2002), 212–218.

[14] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Proceedings of TPCTC*. Lyon, France, 237–252.

[15] Stefano Rizzi, Matteo Golfarelli, and Simone Graziani. 2015. An OLAM Operator for Multi-Dimensional Shrink. *Int. J. of Data Warehousing and Mining* 11, 3 (2015), 68–97.

[16] Oscar Romero and Alberto Abelló. 2007. On the Need of a Reference Algebra for OLAP. In *Proceedings of DaWaK*. Regensburg, Germany, 99–110.

[17] Sunita Sarawagi. 1999. Explaining Differences in Multidimensional Aggregates. In *Proceedings of VLDB*. Edinburgh, Scotland, 42–53.

[18] Sunita Sarawagi. 2000. User-Adaptive Exploration of Multidimensional Data. In *Proceedings of VLDB*. Cairo, Egypt, 307–316.

[19] Gayatri Sathe and Sunita Sarawagi. 2001. Intelligent Rollups in Multidimensional OLAP Data. In *Proceedings of VLDB*. Roma, Italy, 531–540.

[20] Panos Vassiliadis and Patrick Marcel. 2018. The Road to Highlights is Paved with Good Intentions: Envisioning a Paradigm Shift in OLAP Modeling. In *Proceedings of DOLAP*. Vienna, Austria.

[21] Panos Vassiliadis, Patrick Marcel, and Stefano Rizzi. 2019. Beyond Roll-Up's and Drill-Down's: An Intentional Analytics Model to Reinvent OLAP. *Information Systems* 85 (2019), 68–91.

[22] Leslie Owen Wilson. 2016. Anderson and Krathwohl - Bloom's Taxonomy Revised. thesecondprinciple.com/teaching-essentials/beyond-bloom-cognitive-taxonomy-revised/.

[23] Kanit Wongsuphasawat, Yang Liu, and Jeffrey Heer. 2019. Goals, Process, and Challenges of Exploratory Data Analysis: An Interview Study. *CoRR* abs/1911.00568 (2019).

SolveDB$^+$: SQL-Based Prescriptive Analytics

Laurynas Siksnys, Torben Bach Pedersen, Thomas Dyhre Nielsen, Davide Frazzetto
Department of Computer Science, Aalborg University, Denmark
{ siksnys, tbp, tdn }@cs.aau.dk david.frazzetto@gmail.com

ABSTRACT

Today, advanced data analysts make use of both predictive models and optimization problem solving to build data-driven decision making applications, a combination of technologies recently termed *Prescriptive Analytics* (PA). Current PA applications typically have multiple layers of poorly integrated components: a relational DBMS for data storage/management, ML tools for prediction, and specialized software packages for problem modeling and optimization problem solving. This complex stack leads to inefficient, labor-intensive, and error-prone PA workflows, blocking wider adoption of PA. In this paper, we present SolveDB$^+$ – an RDBMS for PA applications which supports all PA steps with *modeling*, *predictive*, and *optimization* functionalities, and integrates these in a common SQL-based framework. Major SolveDB$^+$ novelties are 1) a powerful SQL-based approach for PA problem specification and solving, 2) an extensible in-DBMS infrastructure for prediction and optimization solvers, and 3) in-DBMS modeling and management of PA models. SolveDB$^+$ significantly improves both PA developer productivity and performance.

1 INTRODUCTION

As the next step after Predictive Analytics, *Prescriptive Analytics* (PA) has recently emerged as a new frontier in analytics, combining data management, predictive analytics and ML, and operations research [17]. PA provide a specific course of action for questions such as "How should we maximize our sales in Europe?" PA systems are still in their infancy, typically glued together in an ad-hoc system with separate analytics and optimization tools on top of an RDBMS. There are no integrated PA platforms that combine *data management*, *predictive*, and *optimization* functionalities using a single language, e.g., the frequently used in-DBMS analytics engines only support the first two.

As a running PA example, we consider renewable energy optimization. In a building, PV panels produce intermittent, varying electricity, to run its Heating, Ventilating, and Air Conditioning (HVAC) system. We want to reduce energy costs by using more PV electricity, which requires aligning HVAC operation to PV supply ahead of time, taking forecasted prices and user comfort into account. Table 1 shows a dataset for this case. Input data is a multivariate time series of outdoor (`OutTemp`)/indoor (`inTemp`) temperatures, HVAC consumption (`hLoad`), and PV production (`pvSupply`) per hour. Rows 07:00 - 11:00 are historical data from sensors. Rows 12:00 - 16:00 define future states: `outTemp` contains forecasted outside temperatures; the unknown values of `inTemp`, `hLoad` and `pvSupply` in 12:00 - 16:00 represent decision variables for which PA should compute values by aligning `hLoad` with `pvSupply` at the next 5 hours such that `inTemp` remains within the 20–24°C comfort range and HVAC power limits (0–17kW) are respected. The workflow below exemplifies the 5 overall phases of PA seen in Figure 1.

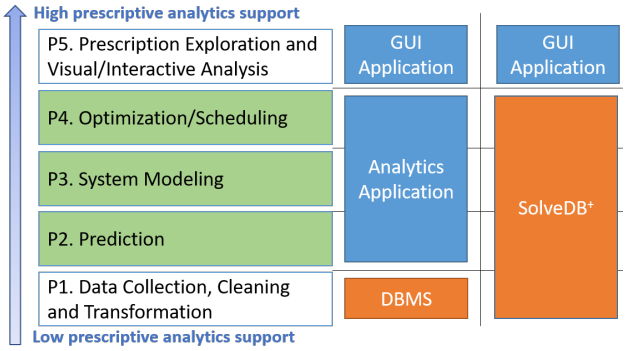


Figure 1: The 5 PA phases and the used software stacks

P1: Collect, clean, validate, and transform the input data.
P2: Predict PV supply given `pvSupply` and `outTemp`.
P3: Model `inTemp` dynamics in relation to `inTemp` and `hLoad`, which requires tuning parameter values specific to this building.
P4: Find optimal `hLoad` values by minimizing electricity cost subject to initial conditions, `pvSupply`, `hLoad`, and comfort constraints, applied over the calibrated model (P3).
P5: Analyze, visualize, and validate the results.

Traditionally, this PA workflow requires a complex software stack with different tools for data management, forecasting, system modeling, and optimization, leading to several *problems*: **Steep Learning Curve:** Different tools have different usage and modeling methodologies, making the learning curve for building PA applications much steeper, which, in turn, leads to more *errors* and *misuse*. **Poor Developer Productivity:** The tools are based on different programming/query languages and have to be glued together in ad-hoc ways to realize PA workflows, leading to *poor developer productivity*, *tool incompatibilities*, and even more *errors* [2]. **Bad performance:** Large amounts of data have to be shipped back and forth between the many tools, leading to high *I/O and memory costs* and *long runtimes* (see Sec. 5). To remedy these problems,these *research challenges* (RCs) must be met:
RC1: Provide a concise yet powerful SQL-based syntax for PA decision problems, supporting efficient query processing.

Table 1: Input dataset for campus energy management.

time	outTemp	inTemp	hLoad	pvSupply
2017/07/02 07:00	05	21	100	0
2017/07/02 08:00	06	20.5	250	0
2017/07/02 09:00	06	21	150	200
2017/07/02 10:00	07	23	120	254
2017/07/02 11:00	08	23	80	320
2017/07/02 12:00	09	?	?	?
2017/07/02 13:00	11	?	?	?
2017/07/02 14:00	12	?	?	?
2017/07/02 15:00	11	?	?	?
2017/07/02 16:00	11	?	?	?

RC2: Provide a concise yet powerful way to share optimization models across sub-problems of the overall PA problem.
RC3: Provide a powerful, easy-to-use, and extensible way of transparently integrating external prediction functionality into PA workflows.
RC4: Seamlessly integrate RC1–RC3 in a SQL-based system.

To meet these challenges, we present SolveDB$^+$. The fact that most PA systems use an RDBMS for data storage [16, 17] combined with the huge popularity of in-DBMS analytics (see Sec. 2), motivates us to propose the first SQL-based in-DBMS platform for PA applications, with these features (www.daisy.aau.dk/solvedb):

Supporting all PA phases: SolveDB$^+$ integrates data management, prediction, system modeling, and optimization in a single tool, yielding better PA productivity. **Extensibility:** SolveDB$^+$ allows developers to add new functionalities for custom PA applications. **Unified SQL-Based PA language:** SolveDB$^+$ extends SQL with new declarative constructs for unified PA problem modeling and analytical functionalities. An entire PA workflow, including forecasting, simulation, and optimization models, can be expressed in a single extended SQL query. **High performance:** The built-in PA algorithms (and user extensions) run in-DBMS, yielding more efficient execution and data exchange. Our experiments show that SolveDB$^+$ yields up to three orders of magnitude better performance for individual PA steps, and up to 3.5 times faster execution and 3 times smaller implementations for complete PA workflows, compared to state-of-the-art baselines, thus combining performance with usability/productivity.

The remainder of the paper is structured as follows. Section 2 discusses related work. Section 3 describes SolveDB$^+$'s prediction framework. Section 4 presents its new PA problem modeling features. Section 5 provides the experimental evaluation. Finally, Section 6 concludes and points out future work.

2 RELATED WORK

A recent extensive survey [16] identifies major emerging trends, remaining challenges, and available technology in the field of PA. In the classification used in this survey, SolveDB$^+$ falls in the category of analytical DBMSes, where analytical functionality is integrated directly within the DBMS back-end. Efforts within this category can be classified into *prediction* DBMSes, for forecasting and probabilistic analysis, and *optimization* DBMSes, for optimization problem solving. Table 2 summarizes and compares essential relevant systems in these sub-categories. The systems are compared in terms of: 1. *What primary language is used for data management* (Data QL); 2. *What primary language is used to specify analytics (incl., prediction and optimization) tasks* (Anl. QL); 3. *Does the system offer native support for predictions?* (Pred); 4. *Does the system offer native support for physical system models and estimating their parameters?* (Est), 5. *Does the system support optimization problem solving?* (Opt); 6. *Does the system support optimization (sub-)models that can be stored natively and manipulated as first-class citizens in the database, and re-used to forms more complex models?* (Mod). We now review these systems.

In-DBMS analytics is a major trend. Among prediction DBMSes, forecasting and in-database ML is supported by the major commercial DBMSes, Oracle [27], SQL Server [32], and TeraData [13]. Recently, HyPEr [12] and DB4ML [10] provide in-DBMS ML for main memory DBMSes. These systems provide efficient in-DBMS forecasting/ML functions, but lack automatic forecasting model selection, parameter estimation, optimization problem solving, and model management, unlike SolveDB$^+$. The

Table 2: Comparison between relevant tools and SolveDB$^+$

System	Data QL	Anl. QL	Pred	Est	Opt	Mod
Oracle	SQL	SQL	✓	✗	✗	✗
SQL Server	SQL	SQL	✓	✗	✗	✗
TeraData[13]	SQL	SQL	✓	✗	✗	✗
DB4ML[10]	SQL	SQL	✓	✗	✗	✗
HyPer[12]	SQL	SQL	✓	✗	✗	✗
MADlib[5]	SQL	SQL+UDF	✓	✗	✗	✗
F^{2}DB[15]	SQL	ext.SQL	✓	✗	✗	✗
SystemML[1]	R-like	R-like	✓	✗	✗	✗
MLbase[14]	R-like	R-like	✓	✗	✗	✗
SciDB [11]	SQL-like	SQL-like	✓	✗	✗	✗
pgFMU[28]	SQL	SQL+UDF	✓	✓	✗	✗
Searchlight[19]	SQL-like	SQL-like	✗	✗	✗	✗
PaQL[8]	ext.SQL	ext.SQL	✗	✗	✗	✗
InezDB[21]	ext.OCaml	ext.OCaml	✗	✗	✓	✗
Tiresias[23]	SQL	ext.Datalog	✗	✗	✓	✗
LogicBlox[6]	LogiQL	LogiQL	✓	✗	✓	✗
SolveDB[31]	SQL	ext.SQL	✗	✗	✓	✗
SolveDB$^+$	SQL	ext.SQL	✓	✓	✓	✓

open source alternative MADlib [5] extends PostgreSQL with UDFs specialized for ML tasks like clustering, classification, and forecasting. Similar to MADlib, pgFMU [28] offers PostgreSQL UDFs for in-DBMS simulation and parameter estimation of *Functional Mock-up Units* (FMUs). These are interoperable simulation models that can define dynamic behaviour of complex physical systems. While FMUs are often used for predictions (P2, see Figure 1), pgFMU does not support including FMUs into user-defined optimization problems (P4). In comparison, SolveDB$^+$ supports (less detailed) so-called *grey-box* models that can be both simulated and optimized in the same environment. Among stand-alone DBMSes, F^{2}DB [15] focuses on time series forecasting in an SQL-based environment. While F^{2}DB specializes in, and is highly optimized for, time series forecasting tasks and employing specific model reuse and maintenance techniques, it does not support the development and integration of user-defined "do-it-yourself" models and generic library models, unlike SolveDB$^+$. In the Big Data context, systems such as SystemML [1], MLBase [14], and SciDB [11] integrate general-purpose declarative machine learning tools that offer scalable distributed computations. In the context of PA, *all* systems (except pgFMU) in this category *only offer support for the predictive analytics phase (P2).*

The optimization DBMSes have focused on advanced what-if scenarios, in-DBMS optimization problem solving, and search under advanced forms of constraints. Systems such as Searchlight [19] and PaQL [8] exploit powerful constraint solvers when processing advanced data search queries. InezDB [21] proposes a formal logic for the symbolic manipulation of optimization models inside a DBMS. Tiresias [23] and LogicBlox [6] provide users a Datalog-based language for what-if scenario analysis. Being the predecessor of SolveDB$^+$, SolveDB [31] is an extension of PostgreSQL for in-DBMS optimization problem solving and solver integration. SolveDB$^+$ extends SolveDB in the directions covering the highlighted PA phases in Figure 1. These new features in SolveDB$^+$, together with their impact (to be observed in Section 5), are highlighted in Table 3. These correspond 1-1 to the research challenges RC1–RC3 mentioned in Section 1, while the integrated SolveDB$^+$ system corresponds to RC4.

Table 3: New features of SolveDB$^+$ compared to SolveDB.

Feature	Description	Impact
In-DBMS Predictive Framework	Specialized forecasting models that are easy to install, (auto)select, and use.	Forecasting easier to use and up to 6 times faster.
Shared Optimization Models	Allow defining reusable optimization (sub-)models stored in-database with their objective functions, constraints, and data specs.	Up to: 2X less code for P3-P4, 16% less code for P1-P4, similar performance.
New Language Features	Asterisk notation, common decision table expressions, model inlining allow specifying PA problems more concisely/efficiently.	Up to 5X less code for P2-P4, similar performance.

In summary, Table 2 shows that while predictive and optimization DBMSes offer some level of in-DBMS analytics support, they do it only for *some* PA phases and do not offer "SQL for all PA phases" like SolveDB$^+$. In comparison, SolveDB$^+$ is the only system to combine and unify predictions and optimization problem solving within a single SQL-based system.

Explainability, also called interpretability, of ML pipelines has received much attention in recent years. It has been considered both for specific categories of ML pipelines, e.g., user group analytics [25] or data exploration [18], and more generally in a survey of AutoML pipelines [33]. In comparison, SolveDB$^+$ focuses on another category, PA pipelines, and supports explainability in PA phases P1-P4. For P1, we do not claim any new contributions, but simply offer the time-honored explainability of SQL. For (external) Prediction methods (P2), we inherit their existing explainability and add to it by declaratively specifying input and output in the solver specs. For System Modeling (P3) and for overall integration of the phases, our high-level declarative SQL-based syntax and shared models allow a higher level of abstraction which is more compact and explainable than a traditional imperative-style ML pipeline. For Optimization (P4), the declarative specifications of objective functions are immediately explainable. Section 5 provides more details.

Another key aspect of ML pipelines is their connectivity to other components/frameworks [33]. As for the "inbound" connectivity, external components are integrated for use in SolveDB$^+$ in two ways. Like other in-DBMS analytics tools (see Tab. 2), SolveDB$^+$ uses UDFs to wrap external functions for direct use in SQL queries, Specifically to SolveDB$^+$, the solver concept is used to integrate external prediction components in a seamless way (see Sec. 3). As for the "outbound" connectivity, SolveDB$^+$ can be integrated in larger pipelines just like other SQL-based in-DBMS analytics tools.

3 PREDICTION

The first phase in Figure 1 *P1: Data Collection, Cleaning, and Transformation* is well supported by the SQL queries, built-in functions, and UDFs of traditional RDBMSes [16], including SolveDB$^+$. Since PA applications need to look ahead in time, effectively supporting the next phase *P2: Prediction* is a key research challenge (RC3). This section describes how we meet RC3. While SolveDB$^+$ can accommodate different models and algorithms for prediction (using both built-in and external tools), it offers dedicated support

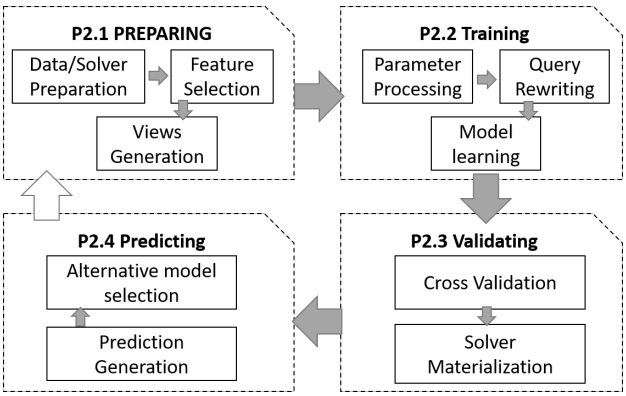


Figure 2: Prediction process + SolveDB$^+$ implementation.

for *time series forecasting* methods. These are widely used for data-driven prediction based on current and historical data.

3.1 Time series forecasting in SolveDB$^+$

Following the energy planning example, the input to the prediction phase is the time series shown in Table 1. The objective is to predict the PV supply for the next 5 hours, by filling in the missing `pvSupply` values in Table 1. This is accomplished by a specific time series forecasting method (e.g., regression) involving a number of steps, as shown in Figure 2: *preparing* – extracting and formatting the data to fit forecasting models, *training* – fitting the forecasting models on the dataset, *validating* – validating the models using cross validation or other evaluation procedures, and *predicting* – forecasting new values.

To support the user in using these methods, SolveDB$^+$ provides its in-DBMS *Predictive Framework*, which (1) exposes various time-series forecasting methods through SQL ("transparently integrating" in RC3), (2) hides the complexity ("easy-to-use" in RC3) of choosing and using these methods (the *preparation*, *training*, *validation*, and *prediction* steps), and (3) offers different extensibility options when a new forecasting method needs to be integrated ("extensible" in RC3). For example, the prediction problem above can be solved in two different ways, using:

Specific forecasting method The following example query invokes the specific forecasting method ARIMA:

```
1 SOLVESELECT t(pvSupply) AS (SELECT * FROM input)
2 USING arima_solver(predictions := 5, time_window := 5,
3                    features := outTemp)
```

To expose the method, SolveDB$^+$ uses the specialized **`SOLVESELECT`** statement (extending the one from SolveDB [31]), to be described in detail in Section 4. It invokes a SolveDB$^+$-native *solver* (*arima_solver*) to derive a so-called *output relation* (a database table) from a so-called *input relation* (**`SELECT * FROM`** `input`) by adding/deleting rows or filling in values in the specified *decision columns*. In this example, the decision column is `pvSupply`, the values of which are requested to be populated by *arima_solver*. The output relation has the same schema as the input relation, but with the `pvSupply` column filled as shown in Table 4. To derive the output relation from the input relation, `arima_solver` additionally takes solver parameters: the number of predictions (*predictions := 5*), the number of time steps to use for training (*time_window:=5*), and the column (*features:=outTemp*) to use as a feature attribute. The solver then performs the steps of *preparation*, *training*, *validation*, and *prediction* (see Figure 2) using the ARIMA model

Table 4: Output of the *Prediction* phase for the example.

time	outTemp	inTemp	hLoad	pvSupply
2017/07/02 07:00	05	21	100	0
2017/07/02 08:00	06	20.5	250	0
2017/07/02 09:00	06	21	150	200
2017/07/02 10:00	07	23	120	254
2017/07/02 11:00	08	23	80	320
2017/07/02 12:00	09	?	?	**200**
2017/07/02 13:00	11	?	?	**220**
2017/07/02 14:00	12	?	?	**260**
2017/07/02 15:00	11	?	?	**140**
2017/07/02 16:00	11	?	?	**0**

trained on data from the input relation with the given parameters. Thus, **SOLVESELECT** allows the user to invoke any specific predictive solver installed in SolveDB$^+$, including solvers for Linear Regression, Logistic Regression, ARIMA, or the powerful *Predictive Advisor* described next. The carefully designed use of the solver ensures the transparency mentioned in RC3.

Predictive Advisor Users can get automated model selection and configuration by using the *Predictive Advisor*, exposed as `predictive_solver`. This solver hides *model selection*, *feature selection*, and *parameter fitting* from the user, and transparently performs *preparation*, *training*, *validation*, and *prediction* and fills in the missing values in the input relation, thus ensuring "easy-of-use" in RC3. Now, the prediction query above can be rewritten as the following simpler query:

```
1 SOLVESELECT t(pvSupply) AS (SELECT * FROM input)
2 USING predictive_solver()
```

The extensibility offered by SolveDB$^+$ also allows for alternative automated predictive frameworks to be integrated as part of the SolveDB$^+$ predictive advisor ("extensible" in RC3).

3.2 Steps of the Predictive Framework

In SolveDB$^+$, the underlying steps of *preparation*, *training*, *validation*, and *prediction* are standardized and their common routines are shared among different forecasting methods, (ensuring "easy-of-use" in RC3.

P2.1 Preparing When the predictive solver (e.g., `arima_solver`) is invoked, the input relation is first analyzed. The framework extracts *decision* (i.e., to be populated with values) and *feature* (to be used as features) columns specified by the user. After recognizing the types of the input columns, it selects candidate solvers from the pool of predictive solvers by comparing the set of decision and features columns to those supported by the solvers. The framework logically partitions the input relation into the training, test, and validation segments by matching the schema for each candidate solver. The selected solver(s) are then used for the training step.

P2.2 Training Next, the model-specific parameters of the candidate solvers are tuned on the training segment of the input relation. The predictive framework automatically generates a **SOLVESELECT** query that specifies an optimization problem with model parameters as decision variables to optimize. This optimization problem is solved by utilizing the solving capabilities of SolveDB$^+$ (Section 4). For example, the ARIMA solver is installed with the standard ARIMA parameters `ar`, `i`, and `ma`, each associated to the domain $[0, 5]$. Therefore, `predictive_solver` described earlier automatically and transparently invokes the following parameter estimation query:

```
1 SOLVESELECT p(ar, i, ma) AS
2   (SELECT NULL::int AS ar,NULL::int AS i,NULL::int AS ma)
3 MINIMIZE(SELECT arima_rmse(
4          ar:=SELECT ar FROM p,
5          i := SELECT i FROM p,
6          ma := SELECT ma FROM p))
7 SUBJECTTO (
8          SELECT 0 <= ar <= 5, 0 <= i <= 5, 0 <= ma <= 5
9          FROM p)
10 USING swarmops.pso()
```

The above **SOLVESELECT** query specifies a global black-box optimization problem, where the values of the parameters `ar`, `i`, and `ma` are found by minimizing the RMSE between the training set and the ARIMA predictions, computed by the function `arima_rmse` in the **MINIMIZE** clause (line 3). The **SUBJECTTO** clause specifies the range in which the parameters can vary. The optimization solver `swarmops` uses a built-in particle swarm optimization method [20] to iteratively attempt to improve a candidate solution with regards to RMSE.

P2.3 Validating Next, the candidate predictive solvers are compared using cross validation. The solver/model leading to the lowest error is selected. As a side effect, the calibrated model instances are stored in a database as user-defined type (UDT) entities for fast reuse of the solver result later.

P2.4 Predicting Finally, predictions are generated by the selected best candidate solver and returned to the user in the form of an output relation of **SOLVESELECT** (Table 4). As **SOLVESELECT** expresses a view over the input relation (Table 1), no user tables are modified in the database.

3.3 Developer Interface

SolveDB$^+$ addresses the "extensible" in RC3 by providing the user with a developer interface to install new in-DBMS predictive solvers. There exists two categories of solvers: *black box* and *white box*. *Black box* solvers are expected to manually handle the steps of data preparation, feature selection, cross-validation, etc., thus *overriding* the predictive framework functionalities. In contrast, *white box* solvers expose the model specifics (e.g., model parameters, their types, etc.) as well as model training and prediction logic to the predictive framework. This way, the solvers may use the functionalities (e.g., **SOLVESELECT**) provided by SolveDB$^+$ for preparing, training, and validating. Such solvers use the solver extensibility capabilities already present in SolveDB [31]. This allows the developers to easily expand the system by taking advantage of existing SolveDB$^+$ solvers/functionality and integrating new prediction models from existing frameworks, e.g., Scikit-Learn [3], Weka [9], MATLAB [22], Statsmodels [29], and TensorFlow[7].

As we will show in Section 5, SolveDB$^+$ is able to offer reduced PA application development efforts and improved overall performance after the integration of desired solvers, yielding up to 5 times more compact problem specifications and up to 6 times reduced forecasting time, compared to SolveDB and commonly used predictive frameworks.

4 OPTIMIZATIONS AND SYSTEM MODELING

Optimization problem solving is essential in 3 of the 5 PA phases (P2, P3, P4), and it therefore plays an essential role in SolveDB$^+$. To deal with optimization problems, SolveDB$^+$ borrows a number of solvers from SolveDB for the different classes of optimization problems, including *linear programming* (LP), *mixed-integer programming* (MIP), and blackbox global optimization (GO), some of

Table 5: LR problem variable layout and a new c_mask column introduced during the CDTE rewrite

id	*pOTemp*	*pMonth*	*pEps*	*error*	**c_mask**
1	*pOTemp*	*pMonth*	*pEps*	e_1	B'11'
2				e_2	B'01'
...				...	B'01'
M				e_M	B'01'

which were already demonstrated in Section 3. To address RC1, SolveDB$^+$ further extends the query syntax used for accessing these solvers. We now elaborate on these new language features.

4.1 Model Specification Syntax

SolveDB$^+$ uses the following syntax to interact with various (e.g., LP/MIP) solvers registered in the active database:

```
1 {SOLVESELECT | SOLVEMODEL}
2         [alias[(col_name[,...])] AS](select_stmt)
3 [INLINE [alias                   AS](select_stmt) [,...]]
4 [WITH   [alias[(col_name[,...])] AS](select_stmt) [,...]]
5 [MINIMIZE (select_stmt) [MAXIMIZE (select_stmt)] |
6  MAXIMIZE (select_stmt) [MINIMIZE (select_stmt)]]
7 [SUBJECTTO [alias AS] (select_stmt) [,...]]
8 [USING solver_name[.method_name][(param[:=expr][,...])]]
```

As shown earlier, the user can use **SOLVESELECT** to define a *model* and pass it to SolveDB$^+$-compliant solver `solver_name` for evaluation using an optionally specified solving method, `method_name`, all defined as follows.

A *problem model* m is defined as a 4-tuple (D, R, s, m). D is the specification of *data and decision variable columns* (lines 2,4). R is the specification of rules that define how the values of the decision variable columns should be instantiated (lines 5-7). s is the name of the solver (`solver_name`) that should evaluate the rules R on the given D using some method m (`method_name`, line 8). Both D and R define two separate sets of specially annotated database relations. Specifically, $D = (D_1^{a_1}, D^{a_2}, \ldots, D_N^{a_N})$ where, $\forall i \in 1 : N, D_i^{a_i} = (c_1, \ldots, c_k, \overline{c_1}, \ldots, \overline{c_l})$ is a SELECT statement (`select_stmt`) defining a database relation with the alias a_i (`alias`) assigned and defined by k *data columns* $c_1, \ldots, c_k$ and l so-called *decision columns* $\overline{c_1}, \ldots, \overline{c_l}$ (`col_name`). Decision columns denote that their rows are decision variables, the values of which should be computed by s. Here, $D_1^{a_1}$ (line 2) is denoted as *input relation*. In a similar way, $R = (R_1^{min}, R_2^{max}, R_3^{a_3}, \ldots, R_M^{a_M})$ is a set of relations that contain s-specific representations of rules defining how decision column values in D should be computed. For convenience, the aliases of R_1^{min} and R_2^{max} are fixed and they are specified in the **MINIMIZE** and **MAXIMIZE** clauses, respectively (line 5-6). The remaining $R_3^{a_3}$ to $R_M^{a_M}$ are specified in the **SUBJECTTO** block along with their respective aliases (line 7). This provides powerful yet concise model specs for RC1.

A *solver* in SolveDB is a user-defined function (UDF) capable of producing (a query for) a so-called *output relation* O in the schema of the *input relation* $D_1^{a_1}$ from a given problem model instance (D, R, s, m) and additionally supplied solver parameters `param` (line 8). SolveDB$^+$ assumes the following standard scoping rules within SOLVESELECT. Each $d_i^{a_i} \in D$ may access a relation $d_j^{a_j} \in D$ using the alias a_j if $j < i$, i.e., $\forall d_i^{a_i} \in D : scope(d_i^{a_i}) = \{(a_j \mapsto d_j^{a_j} | d_j^{a_j} \in D, j < i\}$. Each $r_i^{a_i} \in R$ may access all data and decision variable tables, i.e., $\forall r_i^{a_i} \in R : scope(r_i^{a_i}) = \{(a \mapsto d^a | d^a \in D\}$.

For example, consider a predictive solver (for P2) based on linear regression (LR). In SolveDB$^+$, LR model parameter estimation is specified using the following **SOLVESELECT**:

```
1 SOLVESELECT p(pOTemp, pMonth, pEps) AS (SELECT * FROM pars)
2 WITH e(error) AS (SELECT *, NULL::float8 AS error
3                   FROM input)
4 MINIMIZE (SELECT sum(error) FROM e)
5 SUBJECTTO(SELECT -1*error <=
6   (pOTemp*outTemp + pMonth*month(time) +
7   pEps - pvSupply) <= error FROM e, p)
8 USING solverlp.cbc()
```

Here, lines 1-3 specify model *data and decision columns*. Lines 4-7 specify *rules* that define an objective function and constraints that involve decision variables from the tables p and e. Finally, line 8 specifies *solverlp* and *cbc* as a SolveDB$^+$-compatible solver and a solving method, respectively.

This general **SOLVESELECT** syntax based on standard SQL SELECTs allows exposing different kinds of models and solvers to user queries in a powerful yet concise way (RC1). Compared to SolveDB, SolveDB$^+$ uses a number of novel modeling features unavailable in SolveDB. These are outlined in the remainder of this section.

4.2 Asterisk notation

To support RC1's need for concise and powerful syntax , SolveDB$^+$ proposes the asterisk (*) notation for decision variable column specification (`col_name`). Like SELECT * in the standard SQL, this allows declaring all table columns as decision variables, thus offering more compact problem specifications. Using asterisks, Line 1 in the above optimization problem can be concisely specified as **SOLVESELECT** `p(*)` **AS** (**SELECT** * **FROM** `pars`).

4.3 Common Decision Table Expressions

In SolveDB, the **WITH** clause within **SOLVESELECT** is not supported. Consequently, decision columns (variables) are only allowed in a single (input) relation $D_1^{a_1}$ (i.e., $N = 1$). Therefore, objective and constraint (SELECT) expressions in the **MINIMIZE**/ **MAXIMIZE** and **SUBJECTTO** blocks may become unnecessarily large and complex. Consider the LR model fitting example. This problem uses 2 collections of decision variables: *pOTemp*, *pMonth*, *pEps* as model parameters and e_1, e_2, ..., e_M ($M >> 3$) as prediction errors. One of the most convenient ways to arrange these variables in a single input relation in SolveDB is depicted in Table 5. Here, *pOTemp*, *pMonth*, *pEps* are contained within a single row and e_1, ..., e_M contained within a single column, with many "empty cells" representing *unbound* decision variables. When not referenced within **MINIMIZE**/**MAXIMIZE** and **SUBJECTTO** expressions, such unbound variables are automatically excluded from computations by SolveDB$^+$. Still, referencing *pOTemp*, *pMonth*, *pEps* in the objective and constraint expressions is quite cumbersome - the user is required to supply the predicate **WHERE** `id=1` in all relevant **MINIMIZE**/ **MAXIMIZE**, and **SUBJECTTO** expressions. This makes problem specifications complex and less readable, especially when more than two variable collections are modeled.

Again meeting RC1's need for concise and powerful syntax, SolveDB$^+$ proposes to extend the SOLVESELECT clause with so-called *Common Decision Table Expressions* (CDTEs). As an extension of Common Table Expressions (CTEs, i.e. WITH queries), these allow specifying additional temporary relations, $D_2^{a_2}$, ..., $D_N^{a_N}$, with or without decision columns, where each relation $D_i^{a_i}$ can be accessed from SELECTs of $D_j^{a_j}$, $j > i$, and in the

MINIMIZE/**MAXIMIZE** and **SUBJECTTO** blocks ($R_1^{min}, \ldots, R_M^{ctr_M}$) using the alias a_i. All decision variables of $D_1^{a_1}, \ldots, D_N^{a_N}$ are solved together in a single optimization problem. Note, when the list of the decision columns is empty ($|\{\overline{c} \in D_i^{a_i}\}| = 0$), the CDTE has the semantics of the standard CTE. As demonstrated earlier, CDTEs in SolveDB$^+$ allow conveniently modeling two or more collections of decision variables, unlike SolveDB.

Efficient CDTE query evaluation ("efficient query processing" in RC1): SolveDB$^+$ efficiently evaluates SOLVESELECT queries with CDTEs in two different ways. SolveDB$^+$ either rewrites the CDTEs to a single input relation and standard CTEs, or passes them to a solver for specialized processing. The first approach is preferred, as it is transparent and applicable to all registered SolveDB$^+$ solvers. Here, SolveDB$^+$ first generates a new input relation ($D_1'^{a_1}$) by joining all CDTEs with decision variables and adding a special bit string attribute `c_mask` (see Table 5) to denote CDTEs relevant to specific rows. Then, SolveDB$^+$ generates and processes a new SOLVESELECT *without* decision variables in CDTEs, by using different projections over the new input relation:

```
1 SOLVESELECT l(pOTemp,pMonth,pEps,error) AS
2             (SELECT * FROM input)
3 WITH p AS (SELECT pOTemp, pMonth, pEps FROM l
4            WHERE (c_mask & b'10') <> b'00'),
5      e AS (SELECT error FROM l
6            WHERE (c_mask & b'01') <> b'00')
7 MINIMIZE(SELECT sum(error) FROM e) ...
```

This syntactical extension does not increase the expressive power of SOLVESELECT as the WITH sub-expressions can always be combined into a joint input relation. Instead, CDTEs allow a more intuitive and comcise organization of decision variables in a SOLVESELECT query ("powerful yet concise" in RC1), which is particularly useful when dealing with many auxiliary variables in complex PA cases.

4.4 Shared Models and Model Management

PA applications often build (optimization) models by combining several existing models, e.g. for P3 in our use-case we want to use a generic linear time-invariant *state-space model* (LTI) for capturing temperature dynamics of the HVAC-equipped campus building, and then apply this model in two optimization problems – *LTI model parameter estimation* and *electricity cost optimization* – P3 and P4 in Figure 1. For the first problem, we want to use our input data to estimate the parameters a_1, b_1, and b_2 of the following discrete LTI model for this specific building:

$$x[n+1] = \begin{bmatrix} a_1 \end{bmatrix} \mathbf{x}[n] + \begin{bmatrix} b_1, b_2 \end{bmatrix} \mathbf{u}[n]$$
$$y[n] = \begin{bmatrix} 1 \end{bmatrix} \mathbf{x}[n] + \begin{bmatrix} 0, 0 \end{bmatrix} \mathbf{u}[n]$$

Here, $\mathbf{x}$ is the system 1×1 *state vector* denoting the *inside temperature* of the building; $\mathbf{u}$ is the system 2×1 *input vector* denoting *outside temperature* and applied *HVAC load*, and $\mathbf{y}$ is the 1×1 *output vector* which just "feeds forward" the inside temperature.

In the second problem, we want to use this LTI model with instantiated parameters a_1, b_1, and b_2 inside the cost optimization problem with additionally specified constraints on state variables (inside temperature bounds) and input variables (HVAC power bounds). Obviously, these two problems share the common specification of the generic LTI model (i.e., equations above). However, the LTI model constraints have to be redefined in each of the problems when using SolveDB, as there is no way to reuse them.

Algorithm 1: Problem model instantiation

Input: m - a generic model; Δm - instantiation model
Output: m' - an instantiated model

1 $D \leftarrow \{d^{alias} \in m.D | alias \notin \{alias | d^{alias} \in \Delta m.D\}\} \cup \Delta m.D$
2 $R \leftarrow \{r^{alias} \in m.R | alias \notin \{alias | r^{alias} \in \Delta m.R\}\} \cup \Delta m.R$
3 **return** *(D, R, m.s, m.m)*

To address RC2, SolveDB$^+$ proposes the concept of a *shared problem model*. The shared problem model is a special user-defined data type (UDT), which can be created via the **SOLVEMODEL** clause sharing the same syntax as **SOLVESELECT** (see above). Instead of returning an output relation, this new clause returns the UDT with the complete problem specification inside, i.e., (D, R, s, m). In SolveDB$^+$, such UDTs can be *transformed*, *used in computations*, or *stored* in a database using SolveDB$^+$ queries. The shared LTI model of the building inside temperature can be specified, for example, as:

```
1 SELECT (SOLVEMODEL
2   pars  AS (SELECT 0.0 AS a1,0.0 AS b1,0.0 AS b2)
3 WITH
4   data0 AS (SELECT 21.0 AS inTemp),
5   data  AS (SELECT time,outTemp,inTemp,hLoad FROM input),
6   simul AS (
7     WITH RECURSIVE t(time, x, inTemp) AS (
8       -- Initial data, for step 0
9       SELECT (SELECT min(ts) FROM data) AS time,
10             (SELECT x0 FROM data0) AS x,
11             (SELECT intemp0 FROM data0) AS inTemp
12    UNION ALL
13      -- Computed data, for steps > 0
14      SELECT (SELECT time+interval '1 hour'),
15             (SELECT a1*x+b1*outTemp+b2*hLoad FROM pars),
16              n.inTemp
17      FROM t LEFT JOIN LATERAL
18               (SELECT time, inTemp, outTemp, hLoad
19                FROM data) AS n
20             ON t.time = n.time - interval '1 hour'
21      WHERE (time < (SELECT max(time) FROM data))
22    SELECT time, x, intemp FROM t)))
```

As seen in the example, this model is, essentially, a placeholder with (dummy) relations for LTI model parameters (*pars*), initial values of the state variables (*data*0), and system inputs to be used for model training or predictions (*data*); and relations that represent simulated system states and outputs (*simul*). This model is fairly useless without actual model parameters and data being specified. Therefore, SolveDB proposes 3 specialized "conside yet powerful" operations on shared problem models: *instantiation*, *evaluation*, and *inlining*.

Model instantiation This operation instantiates a (generic) model into a (specific) problem model instance. This is done by allowing the user to redefine the input relation or any other CDTE in the problem model, along with their decision column list. For this, the operator << and another model are used, e.g.,

```
1 SELECT m << (SOLVEMODEL pars(b2) AS
2   (SELECT 0.995 AS a1, 0.001 AS b1, 0.2::float8 AS b2))
3 FROM model
```

In this example, a generic LTI model `m` is first selected from the table `model`. Then, `m` is instantiated using specifications of another model (say Δm) that is generated with **SOLVEMODEL** in the same query. Finally, the instantiation operator << replaces *pars* in m with *pars* in Δm while denoting $\{b2\}$ as a sole decision column with its initial value given in the table. The semantics of this operator is seen in Algorithm 1.

In general, as seen in Algorithm 1, model instantiation allows transferring an input relation, objective functions, constraint expressions, and any other CDTE expression from a *source model* to a *target model*. All entities that cannot be found using an *alias* in the target model are automatically added (instead of replaced) to the target model. This gives the possibility to inject data, different model parameters, objectives, constraints into a generic model.

Model Evaluation This operation allows accessing data from the input relation or any other CDTE inside the model. For this, SolveDB$^+$ introduces a new **MODELEVAL** clause:

```
MODELEVAL ( select_stmt ) IN ( select_stmt )
```

This clause retrieves a model instance by evaluating the 2^{nd} SELECT expression (`select_stmt`), then turns this model into a number of standard CTEs, and finally evaluates the 1^{st} SELECT expression in the context of these CTEs. Thus, the user can retrieve and inspect data specified by the model, e.g.,

```
MODELEVAL (SELECT a1, b1, b2 FROM pars)
       IN (SELECT m FROM model)
```

Model Inlining This operation allows embedding a model instance into another model instance – specified either by SOLVEMODEL or SOLVESELECT. To inline the model, the **INLINE** clause in **SOLVESELECT** or **SOLVEMODEL** is used, e.g.:

```
SOLVESELECT t(a1,b1,b2) AS
            (SELECT 0.5 AS a1,0 AS b1,0.5 AS b2)
     INLINE m AS (SELECT m <<
(SOLVEMODEL params AS (SELECT a1, b1, b2 FROM t)
       WITH  data0 AS (SELECT 25.0::float8 AS inTemp),
              data AS (SELECT * FROM input
              WHERE hload IS NOT NULL ))  FROM model)
MINIMIZE  (SELECT sum((x-inTemp)^2) FROM m_simulation)
SUBJECTTO (SELECT 0<=a1<=1, 0<=b1<=1, 0<=b2<=1 FROM t)
USING swarmops.sa()
```

This query specifies the problem of *least squares* to fit the LTI model parameters a_1, b_1, b_2 to the given data (Table 1). Here, the INLINE clause specifies that this problem depends on the shared problem model m from the table *model*. Before applying m to the outer problem, the model m has to be first instantiated with new LTI model parameters (line 4), a new initial value of the state variable (line 5), and new training dataset (line 6-7). Note, the decision columns (variables) from the outer problem (a1,b1,b2) are passed to the inner model during the instantiation, so their values can be used in computations defined by the inner model. Given this query, SolveDB$^+$ generates a new (outer) problem instance, making all internal model relations ($m.D$, $m.R$) available to the constraint expressions of the outer problem (lines 8-9) using the prefix $m\_$, where m is the assigned model alias (line 3).

The injection of the decision variables through model instantiation is not the only way to interconnect inner and outer problems in SolveDB$^+$. Another way is to declare that some of the inner model relations (CDTEs) contain decision columns. Consider the optimization/scheduling step of the PA process (P4 in Figure 1). To solve the cost minimization problem, SolveDB$^+$ allows defining the following query:

```
1  SOLVESELECT t(hload, iTemp) AS
2        (SELECT time, outTemp, inTemp, hLoad, pvSupply
3         FROM input WHERE hload IS NULL)
4  INLINE m AS (SELECT m << (SOLVEMODEL
5   data AS (SELECT time,outTemp,0 AS inTemp,hLoad FROM t)
6   WITH data0(inTemp) AS (SELECT NULL::float8 AS itemp))
7                FROM model)
8  MINIMIZE (SELECT sum((hload - pvsupply)*0.12) FROM t)
9  SUBJECTTO
10  -- Bind inner and outer problem variables
11  (SELECT t.inTemp = m_simul.x    FROM m_simul, t
12    WHERE t.time   = m_simul.time),
13  -- Initial conditions
14  (SELECT iTemp=20 FROM m_data0),
15   -- Comfort and HP power constraints
16  (SELECT 20<=intemp<=25, 0<=t.hpload<=17000 FROM t)
17 USING solverlp.cbc();
```

As seen here, model instantiation is used to declare that the attribute `inTemp` in the CDTE `data0` of the model m should be treated as decision column (line 6). Thus, a new decision variable(-s) will be introduced in the inner problem and made available to the specification of the outer problem (line 14).

Algorithm 2 elaborates the semantics of this **INLINE** clause. As seen in the algorithm, SolveDB$^+$ imports the input relation, CDTEs, and rule expressions from the inner model m into the outer model o. Each such expression receives a new prefixed alias for use in the outer problem to prevent naming collisions (lines 3,7). Further, table access scopes of these expressions are reworked such that the new relations (with new aliases) in the outer model can be accessed from the inner model expressions using the initial aliases, and without the need to modify the actual expressions (lines 5,9). In SolveDB$^+$, this is done by introducing additional CTEs in inner model expressions, e.g., **WITH** `data0` **AS** (**SELECT** * **FROM** `m_data0`), where m_data0 becomes a part of the outer model, but $data0$ is used in the inner model instead.

Algorithm 2: Problem model inlining

Input: o - a model instance before inlining; m - a model instance to be inlined; ma - a model alias;
Output: o' - a model instance after inlining

1 $prefix \leftarrow ma +' \_'$;
2 **for** $i \leftarrow 1 : |m_i.D|$ **do**
3 $\quad d^{prefix+a} \leftarrow \{d_i^a | d_i^a \in m.D\}$;
4 $\quad o.D \leftarrow o.D \cup \{d^{prefix+a}\}$;
5 $\quad scope(d^{prefix+a}) \leftarrow \{a_j \mapsto d^{prefix+a_j} | d_j^{a_j} \in m.D, j < i, d^{prefix+a_j} \in o.D\}$;
6 **for** $i \leftarrow 1 : |m_i.R|$ **do**
7 $\quad r^{prefix+a} \leftarrow \{r_i^a | r_i^a \in m.R\}$;
8 $\quad o.R \leftarrow o.R \cup \{r^{prefix+a}\}$;
9 $\quad scope(r^{prefix+a}) \leftarrow \{a_j \mapsto d^{prefix+a_j} | d_j^{a_j} \in m.D, d^{prefix+a_j} \in o.D\}$;
10 **return** *(o.D, o.R, o.s, o.m)*

Finally, as seen above, SolveDB$^+$ can "seamlessly integrate" the RC1-RC3 contributions of Sec. 3 and 4 and thus address RC4, allowing the user o specify a complete PA workflow as an extended SQL query. SolveDB$^+$ offers efficient in-DBMS processing by optimally using the DBMS query optimization and execution machinery for processing solver inputs and outputs, allowing for integrated (cache-aware) and optimized processing of PA workflows. The effects of using SolveDB$^+$ and its novel extensions are evaluated next.

5 EXPERIMENTAL EVALUATION

In this section, we first present results from a SolveDB$^+$ usability study involving a group of data scientists. To support the end user claims about SolveDB$^+$, we also evaluated SolveDB$^+$ on two typical PA use-cases from the fields of *energy* and *supply chain management*. Lastly, we used these use-cases to compare SolveDB$^+$ against SolveDB.

Table 6: Strong and Weak Points of SolveDB$^+$

Strong points
"*Syntax very SQL-like, queries feel natural, intuitive from a SQL users perspective. This also makes it **very** easy to pick up for anyone familiar with basic SQL.*"
"*I liked the syntax that makes you feel you are still working inside the database sphere while solving optimization problems without the need to jump between different solutions/languages*"
"*SolveDB+ is still a database system, meaning that it would be possible to to use it even in legacy systems...*"
"*I think SolveDB+ is a great tool! ... For any professionals I see this type of tool as the only tool for fast analytics.*"
"*... great idea and great tool. I have already suggested one of my students to check it out also...I am surprised how easy it was to implement and solve problems - definitely not the last time I will work with SolveDB+*"
"*Seems like a much more streamlined development experience.*"
"*Easy to use in a database-context*"
"*I do think python is more intuitive, but SolveDB+ is very close.*"
"*The simplicity, readability, easy to adapt and learn.*"
"*Fewer lines of code needed to solve the same problem...*"
"*SolveDB+ was faster than MADlib+pIPython*"
Weak points
"*...for some optimization problems, we need to put some "extra" effort to produce a good "representation" of the problem so it that can be handled by SolveDB+ (e.g. Sudoku solver). SolveDB+ needs a big community, and more detailed documentations and examples.*"
"*Needs to be updated on every PostgreSQL release*"
"*Due to relational nature of SQL syntax, some expressions are longer than they ideally should be*"

5.1 Usability Study

We conducted a study where the usability of SolveDB$^+$ was evaluated by a group of highly skilled data scientists, namely the 7 participants of the 2.5 day PhD course *Aspects of Advanced Analytics*, organized by Aalborg University in Dec. 2020. Each participant pre-reported strong competences in SQL, Python, PostgreSQL, and optimization problem solving. The participants used SolveDB$^+$ to solve their chosen subset of five simple optimization problems (Knapsack, production planning, Sudoku, curve fitting, and hypothetical DB deletes/inserts) and two more advanced PA problems (demand and supply balancing, heat-pump power optimization)[30]. In all cases, the initial data and the solution had to be stored in a database. For comparison, the participants had to use another in-DBMS analytics stack of their own choice for solving these problems. They agreed to use the stack based on PostgreSQL, the *PyMathProg* Python library for high-level optimization problem modeling, *PL/Python* language extension for in-DBMS Python programming, and the widely used PostgreSQL extension *MADlib*[5] for in-DBMS machine learning. Afterwards, the participants reflected on their experiences.

The study demonstrated that they solved their chosen problems with approx. 1.5-3.5 times less code and approx. 2 times faster SolveDB runtimes when using SolveDB$^+$. They identified a number of strong and weak points of SolveDB$^+$ - see Table 6. They also reflected on the new SolveDB$^+$ features, e.g., "*The SolveDB+ shared model concept is interesting...*", "*I think it [shared models] fits well with the rest of the system, ... can be incredibly useful in specific use cases...*" , "*...it is a great idea to incorporate the opportunity to do simulation models within the dbms... however, when doing this, my experience is that I need a lot of flexibility - and im not sure the compact style of solveDB+ will benefit me there. At least not yet*". In summary, the study confirmed our expectations that SolveDB$^+$ has good usability, explainability, developer productivity, and performance, even for new users. The next subsections dig deeper into these aspects.

5.2 Experimental Setup

To support the claims about SolveDB$^+$ (Section 5.1), we further evaluated SolveDB$^+$ in two typical PA use-cases from the fields of *energy* and *supply chain management*, covering the phases P1-P4 shown in Figure 1. For both use-cases, we implemented two PA technology stacks: 1) a stack consisting of a standard DBMS and relevant state-of-the-art PA tools and 2) a SolveDB$^+$ stack with a number of standard and specialized built-in solvers (used in place of the PA tools). In both configurations, input data is read from the database and the solution is stored back to the database. We compared these two technology stacks by measuring the Effective Lines of Code (eLOC)[24] (relevant since we are comparing high-level languages and eLoc is used in similar comparisons [28, 31]) of the full implementations and their inherent P1-P4 parts. We also compared them in terms of *execution time*, by encompassing database I/O time as well as prediction, model fitting, and optimization problem solving time. Lastly, these use-cases were used to compare SolveDB$^+$ against SolveDB by evaluating novel SolveDB$^+$ features, including *CDTEs*, *shared models*, and the *predictive framework*. In all experiments, we used SolveDB$^+$/SolveDB on top of PostgreSQL 11.2 in the default configuration and native SolveDB solvers for LP/MIP/Blackbox problems [31].

5.3 Energy Planning (UC1)

We evaluated the impact of using SolveDB$^+$ to solve the energy planning problem from the running example, denoted as UC1, using the NIST dataset [4] – containing 8737 hourly aggregates from PV, HVAC, temperature sensors, all from a high precision lab-home. We compared with two different PA technology stacks using either *specialized tools* or *general modeling tools*.

Specialized tools Here, we used standard PostgreSQL, Matlab R2015b, and three powerful specialized libraries, *Statistics and Machine Learning Toolbox*, *System Identification Toolbox*, and *Multi-Parametric Toolbox* (MPT), for *Linear Regression (LR) forecasting*, *state-space (SS) model fitting*, and *dynamical system optimization*, respectively. Specifically, we used a Matlab implementation that uses the following native library functions: `fitlm` to estimate the LR model coefficients, `predict` to produce PV supply forecasts, and `ssest` to fit HVAC state-space model parameters to the given data. The implementation uses the outputs of these functions to define an MPC (model-predictive control) controller with a number of constraints on the system input and state variables and the PV supply amounts used as a reference for minimizing electricity cost. The size of this implementation in eLOC is given in Figure 3(a) as **`Matlab-native`**. As this configuration is the most comprehensive, it is used as a *reference* for this comparison.

General-purpose modeling tools In this configuration, we utilized a standard DBMS, Matlab R2015b, and YALMIP – a Matlab toolbox for rapid prototyping of optimization problems. Like SolveDB$^+$, YALMIP is provided with a variety of solvers for different problem classes. By using both YALMIP and SolveDB$^+$, we modeled *LR model estimation* (P2), *state-space model fitting*

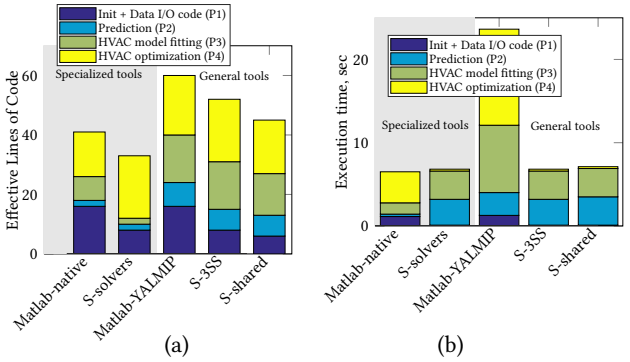


Figure 3: Implementation sizes (a) and run time (b) of UC1

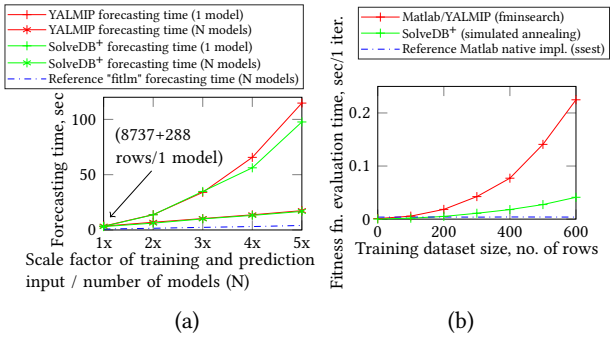


Figure 4: Scalability of prediction (P2) and model fitting (P3) using general-purpose (YALMIP, SolveDB$^+$) and specialized tools (P2: fitlm, P3: ssest)

(P3), and *dynamical system optimization* (P4) problems as explicit LP/nonlinear optimization problems using Matlab/YALMIP programs and SolveDB$^+$ queries, respectively. Specifically, P2 is modeled as an LP optimization problem by minimizing the forecasting error to compute regression model parameters. To solve this problem, SolveDB$^+$ and YALMIP use the Coin-OR CBC solver for the actual computations. Similarly, P3 is specified as a non-linear problem (NLP) of minimizing prediction error of a linear dynamical system using time domain data and *HVAC power levels* and *inside temperatures* as decision variables. To solve this problem, Matlab/YALMIP uses *fminsearch* and SolveDB$^+$ uses *simulated annealing*. These are two distinct NLP solvers that solve the problem in a non-deterministic way. Since they typically give different solutions each time, we only measure average time required for a single solving iteration (fitness function evaluation, Figure 4(b)). Lastly, P4 is modeled as a linear cost minimization problem, where the cost of electricity is minimized under a number of constraints on the HVAC system state and input, and by taking PV supply forecasts into account (based on the LR model). SolveDB$^+$ and YALMIP use CBC to solve this problem. The size of YALMIP implementation in eLOC is given in Figure 3(a) as **`Matlab-YALMIP`**. In SolveDB$^+$, the complete PA workflow, encompassing P2-P4, were implemented in 3 different ways:

-**`S-3SS`** P2-P4 were implemented as three independent **`SOLVESELECT`**s linked using temporary tables (P1).

-**`S-shared`** To be able to reuse the HVAC model parts repeating in P3 and P4, we defined the complete PA problem as a single **`SOLVESELECT`** using a SolveDB$^+$ *shared model*. The model captures indoor temperature dynamics, with P2 and P3 **`SOLVESELECT`** specifications embedded into the model. Note, the size of the model is equally shared by the respective parts in Figure 3(a).

-**`S-solvers`** To relieve the user from the need to specify detailed **`SOLVESELECT`** queries for P2 and P3, we implemented two *composite solvers* which hide respective problem specification details. As these solvers are conceptually similar to the library functions (`Matlab-native`), the overall PA workflow is simplified to a single **`SOLVESELECT`** invoking the composite solvers.

Comparison to specialized tools As seen in Figure 3, the complete PA problem can be specified in just 41 lines of Matlab code and solved in 6.5 secs using specialized tools (`Matlab-native`). Here, around 40% of code and 18% of time is used for initializing libraries and accessing the database, the rest is spent on forming required inputs for, and invoking, the black-box library functions (all considered as P1). As seen for `S-solvers`, this I/O overhead as well as optimization time can be reduced by more than one order of magnitude if all computations are pushed inside the DBMS. This also reduces the PA problem (code) size when SolveDB$^+$ with specialized (composite) solvers are used. As seen in Figure 5 (SolveDB$^+$ vs. MPT), this optimization (P4) performance improvement comes from the reduced model generation time – time spent by MPT to translate the problem to YALMIP, and for YALMIP to aggregate problem constraints and build an optimization (P4) model instance in the binary representation (required by CBC). However, as seen in Figure 3(b), native prediction (P2) and SS model fitting (P3) functions are hard to outperform using general-purpose solvers (`Matlab-native` v.s. `S-solvers`). Figure **??** hint that specialized SolveDB$^+$ solvers for prediction and model fitting are required for larger input datasets. Considering the prediction alone, LR model fitting (P2) using the general-purpose solvers scale *linearly* with respect to independent model count and *exponentially* with respect to training and prediction input size, and therefore might still be useful for some smaller PA cases.

Comparison to general-purpose tools Compared to the native tools (`Matlab-native`), general modeling tools (`Matlab-YALMIP`, `S-3SS` and `S-shared` – all using general-purpose solvers) offer a single language and the full control of how the three PA subproblems P2-P4 are specified. However, explicitly specifying these sub-problems requires up to 45% more code (see Figure 3(a)). Further, computations are up to 3.6 times slower (see Figure 3(b)) and they do not scale (linearly) as in the native case (see Figures 4–5). Comparing YALMIP to SolveDB$^+$, SolveDB$^+$ solves the complete PA problem 3.5 times faster due to significantly reduced data I/O and HVAC optimization time. This can also be seen in Figure 5, which shows that SolveDB$^+$ exhibits up to 2 order of magnitude less data I/O and up to 3 orders of magnitude less model generation time, which is spent translating high-level constraint and objective function specifications into the binary format required by CBC. Both YALMIP and SolveDB$^+$ exhibit somewhat comparable forecasting (P2) and model fitting performance (P3). In the P2 case, YALMIP model generation time is less significant as model constraints can be vectorized (defined without "'for"' loops) and, in the P3 case, just 3 decision variables (a_1, b_1, and b_2) are used. Still, as shown in Figure 4(a), SolveDB$^+$ implementation offers up to 18% lower forecasting time for larger input dataset due to more efficient processing of linear constraints. This difference is less evident when several independent forecasting models need to be estimated using smaller training datasets. Lastly, in addition to these performance benefits, SolveDB$^+$ offers up to 33% smaller implementation sizes as shown in Figure 3(a).

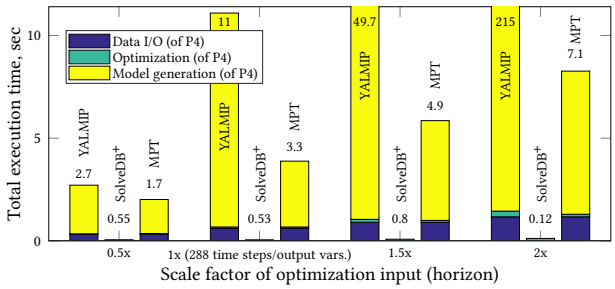


Figure 5: Scalability of HVAC energy optimization (P4)

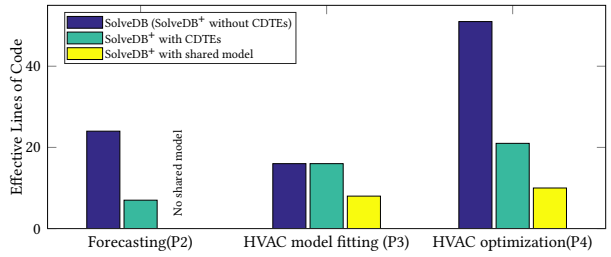


Figure 6: Sizes of SolveDB$^+$ implementations with and without CDTEs and Shared Models

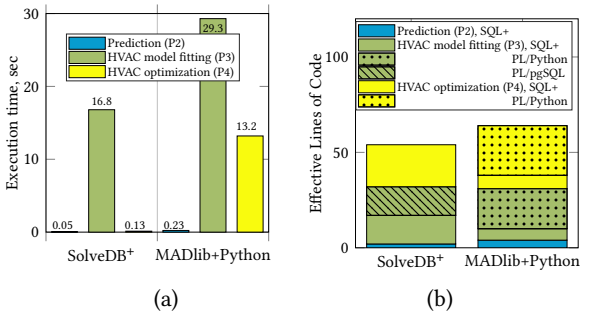


Figure 7: UC1 performance (a) and implementation sizes (b) when using SolveDB$^+$ and existing in-DBMS tools

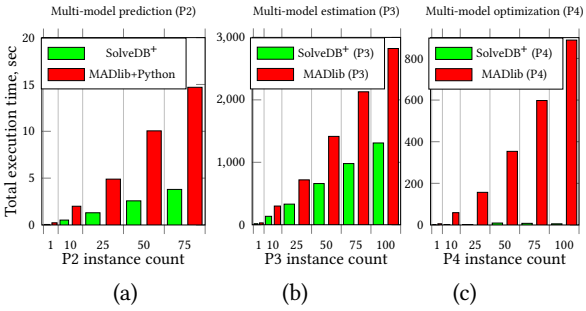


Figure 8: Scalability of In-DBMS UC1 implementations

Comparison to in-DBMS analytics tools Next, we compared SolveDB$^+$ against the in-DBMS analytics stack from the usability study (Section 5.1). We used MADlib's in-DBMS *linear regression* (*linregr_train* UDF) for P2. Since MADlib alone cannot be used to solve the HVAC model fitting and optimization sub-problems (P3-4), we implemented two in-DBMS Python (*PL/Python*) programs for HVAC model fitting (P3) and HVAC operation optimization (P4) by utilizing the *Swarmops* and *PyMathProg* Python libraries, respectively. These libraries offer high-level optimization problem modeling capabilities (required for P3-4) and, under the hood, invoke the low level solvers *Differential Evolution* and *GLPK*, respectively. A SolveDB$^+$ implementation uses three **SOLVESELECT** statements that define the P2-P4 sub-problems and invoke the (same) *linear regression*, *Swarmops*, *GLPK* low-level solvers using SolveDB$^+$'s high-level solvers (incl., *solverlp* and *swarmops* – see Section 3.2 and Section 4.1). The SolveDB$^+$ implementation also uses a *PL/pgSQL* UDF to compute prediction error (being minimized) given (solver-)supplied candidate values of the HVAC model parameters (P3). The goal of this experiment was to compare implementation sizes and runtimes of individual phases (P2-P4) when solving a number of UC1 instances using the same set of low-level solvers (i.e., linear regression, differential evolution, GLPK) running inside a DBMS. Thus, we aimed at comparing the two stacks in terms of *how P2-P4 are specified by the user, how well these (high-level) problem specifications are translated to (low-level) solver inputs*, and *how fast data, solver inputs and outputs are processed by the two in-DBMS stacks.*

As seen in Figure 7(b), *MADlib+Python* required 64 eLOC of mixed SQL and PL/Python code and SolveDB$^+$ required 47 eLOC of (extended) SQL and PL/pgSQL code. While implementation sizes are somewhat comparable, SolveDB$^+$ required very little non-SQL code (15 lines of PL/pgSQL only) to specify the iterative P3 computations. Note, we have also implemented UC1 using pure (extended) SQL (in total 42 lines) with a recursive CTE query for P3. However, this implementation with a recursive query for HVAC simulations might be less intuitive for inexperienced users. In terms of performance, as seen in Figure 7(a), a single instance of UC1 can be solved with SolveDB$^+$ more than twice as fast as with MADlib+Python (19.9 vs 42.7sec). Here, significant gains are observed primarily for P3 (16.8 vs 29.3sec) and P4 (0.13 vs 13.2 sec). For P3, SwarmOPS (in C++) was able to reevaluate the fitness function specified as a SELECT expression from **SOLVESELECT** (that calls a `PL/pgSQL` function) approx. 1.7 time faster that pure Python implementation, where both the solver (SwarmOPS) and the fitness function were implemented in Python. For P4, SolveDB$^+$ offers faster processing of P4 problem symbolic descriptors (*solverlp* vs *PyMathProg*), to be consumed by the same low-level solver (GLPK in C). As seen in Figure 8 (a-c), this gain is more significant when scaling the number of UC1 instances to be solved, i.e., scaling the number of parameters need to be estimated for P3, and predictions and optimization (P2, P4) need to be made for multiple independent HVAC installations. Here, SolveDB$^+$ offered 3.6x faster predictions (P2, Figure 8(a)) since it did not need to create intermediate tables for model parameters and summaries, unlike MADlib; 2.1x faster model parameter estimation, primarily, due to faster evaluation of the fitness function (P3, Figure 8(b)); and 161x faster optimization (P4, Figure 8(c)) primarily due to efficient manipulation of symbolic optimization models and automatic problem partitioning. All in all, SolveDB$^+$ had 2.8x faster execution of the complete PA workflow using less and less complex code, showing its clear advantage over MADlib+Python and confirming the claims about SolveDB$^+$ usability (and performance, see Section 5.1).

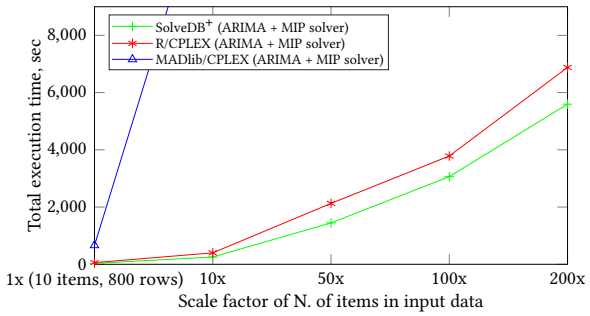


Figure 9: Scalability of combined P1-P4 for UC2

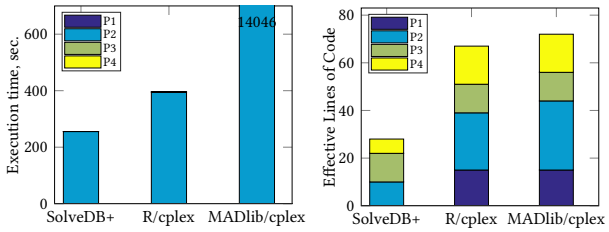


Figure 10: UC2 performance (a) and eLOC (b) comparison

5.4 Supply Chain Management (UC2)

As a second use case (UC2), we considered a common supply chain management scenario. We used the TPC-H dataset [26] containing production supply chain items with the information associated to these items, e.g., orders in the last months, parts needed to assemble the items, size of the parts, price, suppliers, etc. The objective in this use case is to increase revenue by producing in advance the items that will be the most profitable in the next month. The warehouse of the production facility has a limited *volume capacity*, so the decision on which items to produce and store has to be optimized subject to this constraint. This PA workflow requires predicting expected item demand for the next month (P2), modeling *expected profit* for the items by weighting item profit by the probability that the item is ordered in the next month (P3), and solving a variant of the *knapsack problem*, where the warehouse's capacity constraint is respected (P4).

We compared PA stacks with SolveDB$^+$ and both *standalone* and *integrated DBMS analytics* tools. For SolveDB$^+$, we used the predictive framework with a built-in ARIMA solver based on the Statsmodels 0.8.0 package [29] for P2, PL/pgSQL function for P3, and a pre-installed MIP solver from the GNU Linear Programming Kit (GLPK) v4.47 for P4. For standalone tools, we used a configuration with a standard PostgreSQL 9.6.1 (P1, P3), an ARIMA model in R 3.2.3 (P2), and a MIP solver in CPLEX 12.7.1 (P4). For the integrated DBMS analytics tools, we utilized PostgreSQL 9.6.1(P1, P3) with the MADlib [5] extension for in-DBMS machine learning using SQL (P2), and the same MIP solver in CPLEX 12.7.1 (P4). We used 5 different UC2 sizes, scaling the number of items in the dataset. Each item is associated with a time series containing 80 rows of monthly orders.

Figure 10(a) shows the results on the UC2 instance with 100 items. In all implementations, the prediction process accounted almost exclusively for the total execution time, as up to 10000 ARIMA models are trained: 100 per item in R and MADlib, 10 particles with 10 iterations per item in SolveDB$^+$. However, the SolveDB$^+$ implementation was approximately 30% faster than R, and 2 orders of magnitude faster than MADlib, thanks to the efficient use of particle swarm optimization solver for cross validation of the model parameters. Specifically, MADlib does not provide efficient support for cross-validating the forecasting models (ARIMA), with multiple write/read operations accounting for as much as 60% of the total execution time. Figure 10(b) shows the size for the three implementations (implementation size is identical across instances), with SolveDB$^+$ being approximately 50% smaller than the R/MADlib and CPLEX implementations.

The performance results for the different UC2 instances in Figure 9, together with Figure 10(b), show that SolveDB$^+$ allows for a more compact problem definition and execution times that are between 20% and 30% faster than the R configuration, and orders of magnitude faster than the MADlib setup. SolveDB$^+$ outperforms the other two systems thanks to a reduced number of I/O operations and the use of the native local search solvers for hyper-parameters optimization in the model training phase. All in all, UC2 also confirms the end-user claims about SolveDB$^+$ usability (and performance) (Section 5.1).

5.5 SolveDB$^+$ Feature Evaluation (Comparison to SolveDB)

SolveDB$^+$ inherits features and advantages from SolveDB [31]. Specifically, both offer wider applicability and significantly increased tool productivity and usability (order of magnitude less code), while in most cases providing much (up to > 2 orders of magnitude) better performance than systems such as LogicBlox or Tiresias (seeSection 2). We now evaluated the novel SolveDB$^+$ features that distinguish SolveDB$^+$ from SolveDB using the energy and supply chain management use-cases, UC1 and UC2.

Common Decision Table Expressions (CDTEs) As explained in Section 4, CDTEs extend the SOLVESELECT clause like Common Table Expressions (CTEs) extend the simple SELECT in standard SQL. In contrast to CTEs, CDTEs allow annotating some table attributes as *decision columns*, the values of which are evaluated as part of a (much better organized) single SOLVESELECT problem. As seen in Figure 6, CDTEs have a major impact on SolveDB$^+$ usability. Specifying LR model estimation/prediction problems and HVAC optimization problems from the energy planning use-case without CDTEs (SolveDB) requires up to 3 times more **SOLVESELECT** code compared to using CDTEs (SolveDB$^+$). In this case, the HVAC model fitting problem does not benefit from CDTEs, as it uses just a single collection of decision variables, which can be well arranged in a single table. Our experiments also showed that CDTEs do not introduce significant performance overhead to the overall PA workflow.

Shared Optimization Models As explained in Section 4, shared optimization models allow reusing data, objective, and constraint specifications across several optimization problems. UC1 can benefit from such models, by reducing the amount of **SOLVESELECT** code 2 times (Figure 6) for HVAC model fitting and optimization sub-problems alone, and 16% for the complete PA application (see `S-3SS` and `S-shared` in Figure 3(a)), which also includes the shared model specifications. As can be seen in Figure 3(b), shared models do not introduce significant performance overhead to the overall PA workflow.

Predictive Framework As discussed in Section 4, the predictive framework of SolveDB$^+$ offers two ways to integrate new

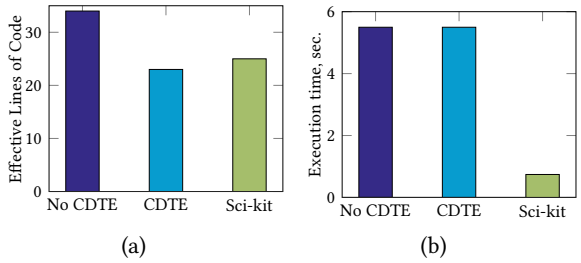


Figure 11: LR code size (a) and execution time (b)

forecasting models. The user can either manually specify forecasting models as SOLVESELECT queries and/or specialized solvers, or install them as "wrappers" over third-party general purpose forecasting libraries. We now compare these two approaches.

For this experiment, we developed the linear regression model as a SOLVESELECT query a) with CDTEs, and b) with no CDTEs wrapped into the respective solvers within the predictive framework. Additionally, we c) installed a general purpose linear regression model from the Sci-kit learn library [3] as a wrapper in SolveDB$^+$. Figure 11(a) shows the implementation size for these three cases. While the size of the Sci-kit implementation is approximately the same as the CDTE implementation, the no CDTE implementation is approximately 30% larger than the other two. Still, the Sci-kit solver implementation is conceptually simpler as it just uses a library function. Furthermore, Figure 11(b) shows that the specialized SolveDB$^+$ implementation is almost 8 times faster than the manual SOLVESELECT implementation (CDTEs do not affect performance), as it combines both in-DBMS execution and a highly specialized machine learning library.

6 CONCLUSION AND FUTURE WORK

This paper presented SolveDB$^+$, the first SQL-based DBMS to provide an extensible and efficient eco-system for all Prescriptive Analytics (PA) phases. SolveDB$^+$ reduces the complexities and inefficiencies of existing PA application stacks, which consist of many specialized, independent, poorly connected systems with different APIs and languages. SolveDB$^+$ acts as a "*swiss-army knife*" system for PA, effectively supporting all 5 phases of PA development: *P1: data management*, *P2: prediction/forecasting*, *P3: system modeling*, *P4: optimization problem solving*, and *P5: solution analysis*. SolveDB$^+$ provides extensibility, allowing developers to add new custom functionalities for specialized PA cases. SolveDB$^+$'s common SQL-based language can express an entire PA workflow in a single SQL-based query. SolveDB$^+$ offers faster PA workflow execution due to its in-DBMS PA algorithms.

Compared to the earlier (SolveDB) tool, SolveDB$^+$ provides a number of novel modeling features, including *common decision table expressions* and *shared optimization models*, enabling a significant size reduction of complex PA problem specifications. It also introduces a new *predictive framework*, which is a generic and extensible in-DBMS platform for the use and development of time series forecasting methods. With all its features, SolveDB$^+$ offers convenient and efficient ways to use and extend the eco-system of forecasting models and optimization problem solvers, thus adapting the system to virtually unlimited PA scenarios.

Our experiments showed that the new SolveDB$^+$ features yield up to 5 times smaller problem specifications (better productivity and explainability) and up to 6 times faster forecasting time, compared to SolveDB. Overall, SolveDB$^+$ offers up to three orders of magnitude better performance for individual PA steps, and up to 3.5 times faster execution times and 3 times smaller implementation sizes for the full PA workflow, compared to state-of-the-art baselines. SolveDB$^+$ scales well in its chosen in-DBMS setting.

Future work will redesign SolveDB$^+$ for distributed Big Data processing and integrate What-If analysis for hypothetical scenarios, and support more data formats, operators on shared models, and further ML models.

REFERENCES

[1] A. Ghoting et al. 2011. SystemML: Declarative machine learning on MapReduce. In *ICDE*.
[2] A. Raj et al. 2020. From Ad-Hoc Data Analytics to DataOps. In *ICSSP*.
[3] F. Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *JMLR* 12 (2011).
[4] H. William et al. 2017. Net Zero Energy Residential Test Facility Instrumented Data; Year 2. (2017). https://doi.org/doi.org/10.18434/T46W2X
[5] J. M. Hellerstein et al. 2012. The MADlib analytics library: or MAD skills, the SQL. *PVLDB* 5, 12 (2012).
[6] M. Aref et al. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD*.
[7] M. Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*.
[8] M. Brucato et al. 2016. Scalable package queries in relational database systems. *PVLDB* 9, 7 (2016).
[9] M. Hall et al. 2009. The WEKA data mining software: an update. *SIGKDD Explor.* 11, 1 (2009).
[10] M. Jasny et al. 2020. DB4ML - An In-Memory Database Kernel with Machine Learning Support. In *SIGMOD*.
[11] M. Stonebraker et al. 2013. SciDB: A database management system for applications with complex analytics. *CiS&E* 15, 3 (2013).
[12] M. Schule et al. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems. In *BTW*.
[13] S. Sanda et al. 2019. In-database Distributed Machine Learning: Demonstration using Teradata SQL Engine. *PVLDB* 12(12) (2019).
[14] T. Kraska et al. 2013. MLbase: A Distributed Machine-learning System.. In *CIDR*.
[15] U. Fischer, F. Rosenthal, and W. Lehner. 2012. F2DB: The Flash-Forward Database System. In *ICDE*.
[16] D. Frazzetto, T. D. Nielsen, T. B. Pedersen, and L. Siksnys. 2020. Prescriptive Analytics: A Survey of Emerging Trends And Technologies. *VLDBJ* 28(4) (2020).
[17] Clyde W. Holsapple, Anita Lee-Post, and Ramakrishnan Pakath. 2014. A unified foundation for business analytics. *DSS* 64, C (2014).
[18] K. Hu, D. Orghian, and C. Hidalgo. [n.d.]. DIVE: A Mixed-Initiative System Supporting Integrated Data Exploration Workflows. In *HILDA*.
[19] A. Kalinin, U. Cetintemel, and S. Zdonik. 2015. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB* 8, 10 (2015).
[20] James Kennedy. 2011. Particle swarm optimization. In *Encyclopedia of machine learning*. Springer, 760–766.
[21] P. Manolios, V. Papavasileiou, and M. Riedewald. 2014. Ilp modulo data. In *FMCAD*.
[22] MATLAB. 2020. *MATLAB API for Python.* Available at se.mathworks.com/help/matlab/matlab-engine-for-python.html.
[23] A. Meliou and D. Suciu. 2012. Tiresias: the database oracle for how-to queries. In *SIGMOD*.
[24] E. Morozoff. 2010. Using a line of code metric to understand software rework. *IEEE software* 27, 1 (2010).
[25] B. Omidvar-Tehrani, S. Amer-Yahia, E. Simon, and et al. [n.d.]. UserDEV: A Mixed-Initiative System for User Group Analytics. In *ILDA*.
[26] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record* 29, 4 (2000), 64–71.
[27] Mark Rittman. 2012. *Oracle Business Intelligence 11g Developers Guide.* McGraw-Hill Osborne Media.
[28] O. Rybnytska, L. Siksnys, T. B. Pedersen, and Bijay Neupane. 2020. pgFMU: Integrating Data Management with Physical System Modelling. In *EDBT*.
[29] S. Seabold and J. Perktold. 2010. Statsmodels: Econometric and statistical modeling with python. In *PiSC*.
[30] L. Siksnys. 2020. *Phd Exercises.* Available at https://www.daisy.aau.dk/wp-content/uploads/2020/12/Advanced-Analytics-Exercises.pdf.
[31] Laurynas Šikšnys and Torben Bach Pedersen. 2016. SolveDB: Integrating Optimization Problem Solvers Into SQL Databases. In *Proc. of SSDBM*. 14.
[32] Z. Tang and J. Maclennan. 2005. *Data mining with SQL Server 2005.* Wiley.
[33] I. Xanthopoulos, I. Tsamardinos, V. Christophides, E. Simon, and A. Salinger. 2020. Putting the Human Back in the AutoML Loop. In *ETLMP*.

Multi-Objective Influence Maximization

Shay Gershtein
Tel Aviv University
shayg1@mail.tau.ac.il

Tova Milo
Tel Aviv University
milo@post.tau.ac.il

Brit Youngmann
Tel Aviv University
brity@mail.tau.ac.il

ABSTRACT

Influence Maximization (IM) is the problem of finding a set of influential users in a social network, so that their aggregated influence is maximized. The classic IM problem focuses on the single objective of maximizing *the overall number of influenced users*. While this serves the goal of reaching a large audience, users often have multiple specific sub-populations they would like to reach within a single campaign, and consequently multiple influence maximization objectives. As we show, maximizing the influence over one group may come at the cost of significantly reducing the influence over the others. To address this, we propose IM-Balanced, a system that allows users to explicitly declare the desired balance between the objectives. IM-Balanced employs a refined notion of the classic IM problem, called Multi-Objective IM, where all objectives except one are turned into constraints, and the remaining objective is optimized subject to these constraints. We prove Multi-Objective IM to be harder to approximate than the original IM problem, and correspondingly provide two complementary approximation algorithms, each suiting a different prioritization pertaining to the inherent trade-off between the objectives. In our experiments we compare our solutions both to existing IM algorithms as well as to alternative approaches, demonstrating the advantages of our algorithms.

1 INTRODUCTION

Social networks attracting millions of people, such as Twitter and LinkedIn, have emerged recently as a prominent marketing medium. *Influence Maximization* (IM) is the problem of finding a set of influential network users (termed a *seed-set*), so that their aggregated influence is maximized [23]. IM has a natural application in viral marketing, where companies promote their brands through the word-of-mouth propagation. This has motivated extensive research [7, 26], emphasizing the development of scalable algorithms [20, 33].

The classic IM problem focuses on the single objective of maximizing *the overall number of influenced users*, given a bound on the seed-set size. While this serves the goal of reaching a large audience, IM algorithms may obliviously focus on certain well-connected populations, at the expense of other demographics of interest. Indeed, marketing campaigns often have multiple objectives, and consequently multiple subpopulations they would like to reach within a single campaign. In this paper we refer to the subpopulations of interest as *emphasized groups*, and assume the existence of boolean functions over user profile attributes, which identify these groups. We introduce the Multi-Objective IM problem, which refines the IM problem, handling multiple emphasized groups.

Ideally, one would like to find a seed-set which simultaneously maximizes the influence over all emphasized groups. However, as we demonstrate, maximizing influence over one group may come at the cost of significantly reducing the influence over another group. Hence, we devise a framework enabling users to explicitly specify the desired trade-off. Concretely, our system, called IM-Balanced, allows the user to prioritize the objectives and declare what portion of the influence over specific groups she is willing to compromise, in order to increase influence over the others.

For simplicity of presentation, we initially focus on the case where the user has two (possibly overlapping) emphasized groups, denoted as g_1 and g_2, and she is willing to compromise a certain percentage of the maximal possible influence over one group for an influence increase over the other. We then extend our discussion to multiple groups, and shortly discuss alternative problem definitions.

We illustrate the problem that we study in this paper via the following two examples.

Example 1.1. Consider a government office aiming to spread a message regarding a new vaccination policy, across a social network. The main goal is to reach the largest possible number of users, but at the same time, it is also desirable to maximize the number of reached anti-vaccination users. Here g_1 consists of all users, and g_2 is the group of anti-vaccination users. A standard IM algorithm will maximize the overall influence (g_1), possibly at the expense of not reaching sufficient g_2 members. A partial solution can be found in targeted IM algorithms (e.g., [9]), which maximize the influence over a particular group (here - g_2). But if this (possibly small) group is somewhat socially isolated, the message may not reach a sufficient number of users overall.

Example 1.2. Consider a tech company running a recruitment campaign over a social network, with the goal of hiring both engineers (g_1) and researchers (g_2). Assume that there are far more engineers than researchers, and that the two groups are not strongly connected socially (though some users may belong to both groups). A targeted IM algorithm focusing, e.g., on users belonging to the union of the groups, may fail to reach a sufficiently large fraction of the researchers. On the other hand, a targeted IM focusing on the researchers may result in too few engineers being reached.

In both examples, there is a trade-off between the influence over two groups of interest. One simple solution is to split the budget (i.e., seed-set size) and run two separate (single-objective) targeted IM algorithms. However, it is not clear how to split the seed-set to obtain the desired balance between the objectives. An alternative approach to tackle multi-objective optimization problems is the weighted-sum approach, where the objectives are combined into a single objective. In the IM setting this involves assigning each user a weight depending on the groups(s) to which she belongs (e.g. [26, 31]). A main difficulty in applying this approach is assigning the weights that achieve a desired influence balance [21]. Indeed, as we demonstrate in our experiments, the exploration for the optimal weights results in poor runtime performance.

Another more direct approach to multi-objective optimization problems is the constraints method [12], where all objectives except one are transformed into constraints, and the remaining

objective is optimized subject to these constraints. Our work employs this approach for IM. Concretely, in IM-Balanced users can define the emphasized groups, and specify for each group the fraction of its optimal influence that they are willing to compromise in order to increase influence over other groups. An easily operated UI allows users to view the maximal possible influence for each group (and what influence it entails over other groups), specify the constraints, and view the corresponding derived influence.

Continuing with Example 1.1, if the UI indicates that the overall number of users that can be influenced is rather high, one may be willing to sacrifice a certain amount in order to increase the influence over anti-vaccination users. In Example 1.2, assuming that the company is interested in recruiting a small number of researchers and a larger number of engineers, one can set a constraint on the minimal number of researchers to be informed, and maximize the influence over engineers under this constraint.

Next, we provide a brief overview of our contributions.

Multi-Objective IM. To allow users to balance the objectives we formalize the Multi-Objective IM problem, which extends the IM problem as follows. Given two emphasized groups g_1 and g_2 and a threshold $0 \leq t \leq 1$, we add a requirement that the solution must exceed a t-fraction of the optimal influence over g_2. Then, subject to this constraint, we maximize the influence over g_1. For $t = 0$ one gets a single-objective targeted IM problem solely over g_1 users, whereas for $t = 1$ one gets a single-objective targeted IM solely over g_2 users (Section 3).

Approximation lower bound. We prove that, like IM, Multi-Objective IM is *NP*-hard. We show that when the constraint threshold t is $> (1 - \frac{1}{e})$, then no seed-set satisfying the constraint can be found in PTIME. Moreover, we prove that the $(1 - \frac{1}{e})$-approximation factor for g_1, which is optimal in the (unconstrained) IM problem, is unattainable in our setting. We show however that it can nevertheless be achieved if the constraint imposed on g_2 is also approximated by a $(1 - \frac{1}{e})$ factor. This bound exposes the trade-off between the approximation factor for the g_1 users and the relaxation of the constraint imposed on the g_2 users. We therefore provide two approximation algorithms, each suiting a different prioritization pertaining to this trade-off.

The MOIM algorithm. Our first algorithm is simple yet highly efficient. It follows the budget splitting approach mentioned above, but rather than requiring the user to specify the partition, it derives it by itself. MOIM runs two single-objective targeted IM algorithms, each focusing on a different group, and combines their outputs. It guarantees that the constraint is fully satisfied, while providing a $(1 - \frac{1}{e \cdot (1-t)})$-approximation for the g_1 users, which equals $1 - \frac{1}{e}$ for $t = 0$, but decreases as t increases. A key advantage of MOIM is its modularity: MOIM maintains the properties of its input IM algorithm, carrying over all of its optimizations, and therefore it achieves near linear time performance. Such good performance is critical for scaling successfully to massive networks (Section 4).

The RMOIM algorithm. To get a tighter approximation ratio one needs to compromise on (i) how strictly the constraint is maintained, and (ii) performance. The RMOIM algorithm relaxes the constraint, allowing its approximation by a $(1 - \frac{1}{e})$ factor, achieving in return near optimal approximation ratio for the influence over g_1. RMOIM extends a Linear program (LP) for Maximum Coverage [38], and thus its performance becomes polynomial (but still practical for real-life social networks including tens of thousands users, as our experiments indicate). One point to note is that building the LP assumes knowledge of the optimal influence over the constrained g_2 group. As this value is incomputable in PTIME, we approximate it, and provide worst case guarantees for this as well.

Implementation and Experimental study. We have implemented our algorithms as part of the IM-Balanced system and experimentally compare our algorithms to (targeted) IM algorithm and alternative approaches. We show that while the weighted-sum approach, *when assigned optimal weights*, is able to achieve results of quality close to ours, our algorithms are significantly more efficient. In terms of runtime performance, we show that the quality advantage comes with a reasonable performance cost for MOIM, which scales well for massive networks. For RMOIM the decrease in scalability turns out to be moderate, proving it practical for non-massive networks, while often exceeding worst-case guarantees to satisfy the constraint (Section 6).

A demonstration of IM-Balanced's usability and its suitability to end-to-end employment was presented in [16]. The short paper accompanying the demonstration provides only a brief, high-level description of the system, whereas the present paper provides the theoretical foundations and algorithms underlying the demonstrated system, as well as the experimental study.

For space constraints, all proofs are deferred to our technical report [3].

2 PRELIMINARIES

This section presents the standard IM problem, and introduces the auxiliary problem of *Group-Oriented IM*. Our multi-objective variant of the IM problem is then presented in the next section.

2.1 Influence Maximization

We model a social network as a weighted graph $G{=}(V, E, W)$, where V is the set of nodes and every edge $(u, v){\in}E$ is associated with a weight $W(u, v){\in}[0, 1]$, which models the probability that u will influence v. Given a function $I(\cdot)$ dictating how influence is propagated in the network, the IM problem [23] is defined as follows.

Definition 2.1 (IM [23]). Given a weighted directed graph G and a natural number $k \leq |V|$, find a set O that satisfies: $O = argmax_{\{T:T\subseteq V,|T|=k\}} I(T)$, where $I(T)$ is the expected number of nodes influenced by the seed set T.

Naturally, every node v in a seed set T is influenced by itself, and hence, by definition, T is influenced by T with probability 1. In what follows, we refer to influenced nodes as *covered*.

The function $I(\cdot)$ is defined by the influence propagation model. The majority of existing IM algorithms apply for the two most researched models [7, 20], the Independent Cascade (IC) and the Linear Threshold (LT) models. Both models define the function $I(\cdot)$ as non-negative, submodular and monotonically rising. Our results hold under both models. For simplicity of presentation, in our numeric examples throughout the paper we focus on the LT model.

In the LT model, each node v chooses a threshold $\theta_v{\in}[0, 1]$ uniformly at random, which represents the weighted fraction of v's neighbors that must become covered in order for v to become covered. Given a random choice of thresholds and an initial set of seed nodes, the diffusion process unfolds deterministically in discrete steps: in step t, all nodes that were covered in step

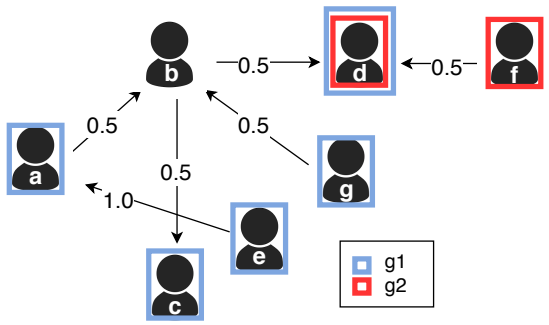


Figure 1: Example social network with two emphasized groups.

$t-1$ remain covered, and we cover any node v for which the total weight of its covered neighbors is at least θ_v. To illustrate, consider the example network presented in Figure 1, ignoring for now the users' border colors. For k=2, the optimal 2-size solution is O=$\{e, g\}$, where $I(O)$=5. Throughout the paper, the threshold for each node was sampled uniformly at random from $[0, 1]$.

Existing IM algorithms. Selecting the optimal seed set is NP-hard, and hard to approximate beyond a factor of $(1-\frac{1}{e})$ [23].The subsequent work on IM following [23], which had already achieved the optimal approximation, has focused on scalability [7, 13, 29]. In what follows, whenever we refer to an IM algorithm, we in fact refer to a probabilistic algorithm, which, given the parameters $0 \leq \epsilon, \delta \leq 1$, achieves, with probability $\geq (1-\delta)$, the optimal approximation factor up to an additive error of ϵ. To ease the presentation, we omit the discussion of ϵ and δ whenever possible.

State-of-the-art IM algorithms are based on the Reverse Influence Sampling (RIS) framework, achieving near optimal time complexity [33] of $\tilde{\Theta}(k \cdot (|V|+|E|))$. The RIS framework utilizes sampling over the transpose graph, to reduce the problem to an instance of the *Maximum Coverage* (MC) problem [38]. For completeness of this paper, we formally define this problem.

Definition 2.2 (MC [38]). Given subsets $S_1, ..., S_m$ of elements from $U = \{u_1, ..., u_n\}$ and a natural number $k \leq m$, the goal is to find k subsets from $S_1, ..., S_m$ so as to maximize the number of covered elements in their union.

The well-known MC problem has a simple greedy approximation procedure [38], achieving an optimal approximation factor of $(1-\frac{1}{e})$. The RIS framework consists of two steps: First, θ nodes are sampled uniformly, then, for each sampled node u, a backward influence propagation is simulated from it, with all nodes covered in a simulation constituting a *Reverse Reachability* (*RR*) set. This *RR* set plays the role of possible influence sources for u. Next, each node is associated with the set of *RR* sets containing it, then, using a greedy algorithm, k nodes are selected with the goal of maximizing the number of covered *RR* sets. The observation underpinning this approach is that influential nodes will appear more frequently in *RR* sets, and that the share of *RR* sets covered by a seed set implies an unbiased estimator for its influence.

Example 2.3. Let k=2, θ=4 and four random *RR* sets G_{d_1}=$\{b, d, f\}$, G_e=$\{e\}$, G_{d_2}=$\{d, f\}$ and G_b=$\{a, b, e\}$ are generated from the graph depicted in Figure 1 (d was sampled twice). The corresponding MC instance is: S_b=$\{G_{d_1}, G_b\}$, S_d=$\{G_{d_1}, G_{d_2}\}$, S_f=$\{G_{d_1}, G_{d_2}\}$, S_e=$\{G_b, G_e\}$, S_a=$\{G_b\}$. W.h.p. the sets S_e, S_f will be selected by the greedy algorithm for MC, as they cover all *RR* sets, and hence the nodes e, f will be selected as the seed nodes.

Most recent works focused on optimizing this approach by minimizing the number of sampled *RR* sets [20, 28, 34].

An important observation is that the second step of RIS can also be achieved using Linear Programming (LP), yielding the same guarantees. However, in terms of time complexity, IM algorithms are nearly linear, compared to PTIME LP solvers [22].

2.2 Group-Oriented IM

In our setting users are associated with profile properties such as their profession or political opinion. Characterized by these properties, the end-user provides her *emphasized groups*, i.e., groups which she wishes to ensure are sufficiently covered. An emphasized group may be defined using a boolean query over (multiple) user profile attributes. Figure 1 depicts two emphasized groups: the group of users with red border (g_1), and the group of users with blue border (g_2). In this example, user d belongs to both groups and user b to none.

Recall that $I(S)$ denotes the expected number of nodes covered by a seed-set S. Let $g \subseteq V$ be a group of emphasized users, and $I_g(S)$ denote the expected number of g members covered by S, referred to as the g-cover. We present the auxiliary *Group-Oriented IM* problem, denoted as IM_g, which instead of maximizing $I(\cdot)$, maximizes $I_g(\cdot)$.

Definition 2.4 (The IM_g problem). Given a group $g \subseteq V$ and a number $k \leq |V|$, find a set O_g satisfying: $O_g = argmax_{\{T:T\subseteq V,|T|=k\}} I_g(T)$.

To illustrate, consider the following example.

Example 2.5. Consider again Figure 1 and assume that k=2. The optimal solution for g_2 is O_{g_2}=$\{d, f\}$, where $I(O_{g_2})$=$I_{g_2}(O_{g_2})$=2 and $I_{g_1}(O_{g_2})$=0. The solution that maximizes the g_1-cover is O_{g_1}=$\{e, g\}$, where $I_{g_1}(O_{g_1})$=4 and $I_{g_2}(O_{g_1})$=0.5. Observe that covering a greater number of users from one group may come at the cost of significantly reducing the cover size of users from another group.

The hardness result of IM also applies to this variant, following a straightforward reduction from IM, where g=V.

Proposition 2.6. *The IM_g problem is hard to approximate beyond a factor of $(1-\frac{1}{e})$ in PTIME.*

In Section 4.1 we explain how a given IM algorithm can be adapted to its group-oriented version, retaining all its theoretical properties. Note that this variant can be seen as a special case of the *Targeted IM* problem [26], where the goal is to maximize influence over a targeted group of users, with relevance of users modeled by weights in $[0, 1]$. The IM_g problem is further imposing a dichotomy where the weights are in $\{0, 1\}$, modeling discrete properties.

3 PROBLEM FORMULATION

As mentioned, our results support multiple, possibly overlapping, emphasized groups. However, for simplicity, we initially focus on the two groups scenario and imposed a size constraint on one group. In Section 5.1 we extend our results to multiple emphasized groups, and discuss alternative problem definitions.

3.1 Multi-Objective IM

Let g_1, g_2 to be two emphasized groups. Our goal is to assure the obtained solution will ensure sufficient cover of the two groups. To this end, we add a constraint on the IM_{g_2} problem (pertaining to the g_2 group), which explicitly models how much the user is willing to settle on the g_2-cover, in order to increase the g_1-cover.

Definition 3.1 (Multi-Objective IM). Given a network G, two emphasized groups $g_1, g_2 \subseteq V$, a threshold parameter $0 \leq t \leq 1$ and a number k, find a k-size seed-set O^* that maximizes the g_1-cover size, subject to the constraint on the g_2-cover being above a t-fraction of its optimal size. Namely, find a set O^* s.t:

$$O^* = argmax_{\{T:|T|=k, I_{g_2}(T) \geq t \cdot I_{g_2}(O_{g_2})\}} I_{g_1}(T)$$

where $I_{g_1}(T)$ (resp., $I_{g_2}(T)$) denote the expected size of the g_1 (resp., g_2) cover by T, and O_{g_2} denotes the optimal k-size solution for g_2.

Throughout the paper, we refer to the expected g_1 and g_2 influences, resp., as the objective and the constraint. To illustrate, in Example 1.1, one may wish to maximize the influence over the anti-vaccination users, while ensuring that the influence over all users is at least 60% of its optimal value. Alternately, continuing with Example 1.2, a user may wish to maximize the influence over engineers, while ensuring that the influence over researchers is no less than 50% of its optimal value.

To illustrate how the constraint affects the selected seed-set, consider the following example.

Example 3.2. Consider again Figure 1 and let $k = 2$. For $t = 0.1$ the optimal solution is $S = \{e, g\}$ since $I_{g_2}(S) = 0.5 \geq 0.1 \cdot I_{g_2}(O_{g_2}) = 0.2$ (O_{g_2} is the optimal solution for g_2), and among all 2-size seed-sets satisfying the constraint, its g_1-cover size is maximal with $I_{g_1}(S) = 4$. However, for $t = 0.5$, S no longer satisfies the constraint, and $S' = \{e, d\}$ becomes the optimal solution, with $I_{g_1}(S') = 3.25$ and $I_{g_2}(S') = 1$. This demonstrates that higher values of t put more emphasis on the g_2-cover, possibly at the expense of eliminating seed-sets with high approximation factor for the g_1-cover.

Recall that the IM problem is closely related to the MC problem, as explained in Section 2.1. We define the *Multi-Objective MC* problem, analogous to Multi-Objective IM, which will serve us for deriving our lower bound and for devising the RMOIM algorithm.

Definition 3.3 (Multi-Objective MC). Given subsets $S_1, \ldots, S_m$ of elements from $U = \{u_1, \ldots, u_n\}$, two groups of elements $g_1, g_2 \subseteq U$, a threshold parameter $0 \leq t \leq 1$, and a number $k \leq m$, a constraint is imposed on the number of covered elements from g_2, requiring it to exceed a t-fraction of the optimal cover size. The goal is to find, among all k sets from $S_1, \ldots, S_m$ satisfying the constraint, the one covering a maximal number of elements belong to g_1.

The constraint threshold. Before presenting our algorithms, let us highlight important properties of the constraint threshold parameter t.

First, consider again Example 3.2, demonstrating that setting higher values for t restricts the solution space and diminishes the optimal value for the objective among remaining k-size seed-set. This exposes the inherent trade-off between the objective and the constraint threshold. A higher threshold is at odds with optimizing the main objective.

We note that the actual value of the optimal g_2-cover size, $I_{g_2}(O_{g_2})$, can only be approximated up to a $(1 - \frac{1}{e})$ factor in PTIME. Thus, the exact value can only be referred to implicitly. Hence, to allow the user to make an informed decision for the value of t, our system uses an IM_g algorithm (as we explain in Section 4), yielding the optimal PTIME approximation for $I_{g_2}(O_{g_2})$.

Observe that setting t to 0 nullifies the constraint, producing the IM_g problem for g_1. Therefore, we only examine cases where $t > 0$. Moreover, it is easy to show that for $t > 1 - \frac{1}{e}$, following the hardness results of IM [23], merely finding a single (not necessarily optimal) k-size seed set satisfying the constraint cannot be done in PTIME.

Corollary 3.4. *A k-size seed set satisfying the constraint can always be found in PTIME only if $0 \leq t \leq (1 - \frac{1}{e})$. For higher t values, this claim no longer holds.*

We therefore restrict our attention to cases where $0 \leq t \leq (1 - \frac{1}{e})$. In cases where the user is interested in higher values of t, as no PTIME algorithm which satisfies the constraint exists, one would need to employ an exhaustive search over the $|V|^k$ possible k-size seed-sets to find the optimal solution.

3.2 Approximation lower bound

In order to devise efficient algorithms for Multi-Objective IM, it is useful to understand which properties are attainable for a PTIME algorithm. We next formally define the solution space, then present a lower bound for Multi-Objective IM.

The solution space. We generalize the solution space to *bicriteria approximation*, where an algorithm approximates the objective and may also approximate the constraint, up to multiplicative factors of α and β, resp. For β=1 the solution strictly satisfies the constraint. To accommodate practical algorithms we consider, as in standard IM, randomized algorithms that may add an error margin ϵ to the approximation factors, while requiring the stated factors to hold with probability $\geq(1-\delta)$. Formally, given $0 \leq \epsilon, \delta \leq 1$, an algorithm computes a (α, β)-solution S, with $0 \leq \alpha, \beta \leq 1$, if for every instance (G, g_1, g_2, k, t) of Multi-Objective IM, the following holds with probability $\geq 1-\delta$: $I_{g_2}(S) \geq (\beta-\epsilon) \cdot t \cdot I_{g_2}(O_{g_2})$ and $I_{g_1}(S) \geq (\alpha-\epsilon) I_{g_1}(O^*)$, where O^* is the optimal constrained solution w.r.t. Def. 3.1. We assume ϵ and δ are implicitly provided. However, for simplicity, we omit discussions of these parameters whenever possible.

We emphasize that α is derived from comparing the returned solution not to the optimal unconstrained solution, but rather to an optimal solution which satisfies the constraint. This highlights the difference between approximating the constraint by a factor of β and replacing t with $\beta \cdot t$, as the solution space is affected only in the latter case. Namely, when examining a seed-set which relaxes the constraint, the optimal value for the objective is still taken only over the subset of solutions satisfying the constraint. We refer to an algorithm *as dominant over another algorithm* if it computes an approximated solution for higher values of at least one parameter (α, β), with the other parameter being at least equal. We refer to a tuple (α, β) as an *optimum*, if no (PTIME) algorithm that generates an approximated solution dominant over it exists. One immediate such optimum is $(1 - \frac{1}{e}, 1)$, which follows directly from the hardness result of IM [23]. However, as we prove, there exists no PTIME algorithm which can achieve this bound. Moreover, we show that to achieve $\alpha = (1 - \frac{1}{e})$, β must be reduced to $(1 - \frac{1}{e})$ as well.

Hardness of approximation. As mentioned, the optimal objective approximation of Multi-Objective IM is α=$1-\frac{1}{e}$. We next prove that in order to achieve this optimal α value, a relaxation of the constraint is necessary. Concretely, we prove that Multi-Objective IM has no PTIME algorithm with approximation guarantees (even in expectation) dominant over $(1 - \frac{1}{e}, 1 - \frac{1}{e})$, via a reduction from MC. This result is independent of t, yet, surprisingly, holds for all its values in $(0, 1 - \frac{1}{e}]$.

Theorem 3.5. *Multi-Objective IM has no approximation factor dominant over* $(1-\frac{1}{e}, 1-\frac{1}{e})$ *(unless NP = BPP).*

Next, we provide a proof sketch for Theorem 3.5 using a novel reduction from MC.

Proof. (sketch). Given an MC instance along with k and t, let k_t denote the smallest natural number s.t. $I(O_{k_t}) \geq t \cdot I(O_k)$. We first fix any arbitrary k and $t \in (0, 1-\frac{1}{e}]$, then sample two disjoint MC instances, $\mathcal{I}_1$ and $\mathcal{I}_2$, s.t. the seed set size requirements are $k-k_t$ and k_t, resp. We construct a Multi-Objective MC instance by taking the union of both collection of sets, and defining the g_1 and g_2 groups as follows: g_1 comprises of all elements of $\mathcal{I}_1$, and g_2 comprises of all elements of $\mathcal{I}_2$. The cardinality constraint is k along with threshold t. This construction implies a dichotomy where choosing sets from the g_1 collection only affects the objective, while choosing sets from the g_2 collection only affects the constraint. We show that, in the worst case, one needs to choose as many g_2 sets as in the optimal solution (i.e. k_t sets), up to a $o(1)$ factor, to achieve a $(1-\frac{1}{e})$ approximation of the constraint, and therefore with the remaining slots one cannot guarantee any factor beyond $(1-\frac{1}{e})$ for the objective.

Last, we extend this result to Multi-Objective IM via a reduction from Multi-Objective MC. In essence, we reduce a given Multi-Objective MC instance to a graph s.t. each element is mapped to a new node, carrying over any membership in g_1 and g_2 groups. Additionally, for each subset S_i, we create a new node, and add an edge from it into every nodes corresponding to an element in this set, with the constant edge weight of 1. □

Note that this lower bound holds even for the easier version of the problem, where explicit values are known for both the constraint threshold and the constrained optimum for the objective.

4 ALGORITHMS

As mentioned, the approximation factor of the objective depends on how strictly the constraint is preserved. We, therefore, provide two complementary algorithms for Multi-Objective IM. Our first algorithm, named *the Multi-Objective IM (MOIM) algorithm*, finds a seed-set that strictly satisfies the constraint, at the cost of influence decrease for the objective. Its key advantage is that it achieves near-linear time complexity, which, as we show, is critical for scaling successfully to massive networks. To get a tighter approximation ratio for the objective, our second algorithm, named *the Relaxed Multi-Objective IM (RMOIM) algorithm*, relaxes the constraint, allowing its approximation by a $(1-\frac{1}{e})$-factor, achieving in return near optimal approximation for the objective. This however comes at the cost of performance - its time complexity is polynomial.

4.1 The MOIM algorithm

MOIM is a simple yet efficient algorithm achieving state-of-the-art performance by leveraging existing IM algorithms. Intuitively, using a modular approach where given an IM algorithm, it generically modifies it to create two group-oriented versions of it, then combines them together to produce a single seed set. We next detail our modification of a given IM algorithm, followed by the full algorithm scheme.

Given an IM algorithm $\mathcal{A}$ and an emphasized group g, we define $\mathcal{A}_g$ as its IM_g counterpart - an analogous algorithm that maximizes $I_g(\cdot)$ instead of $I(\cdot)$. Any RIS-based algorithm, $\mathcal{A}$, can be adapted to $\mathcal{A}_g$ via a single modification: the *RR* sets are generated from nodes from g only, independently and uniformly as before. We can prove that $\mathcal{A}_g$ outputs a seed-set covering at least $(1-\frac{1}{e}) \cdot I_g(O_g)$ nodes from g, which is optimal [23].

A method of weighted RIS sampling for solving Targeted IM was presented in [26]. Concretely, instead of using the uniform distribution, nodes are sampled according to their weights, which model their relevance to a given context. Our adaptation for IM_g can be seen as a special case of this method with binary weights. Nonetheless, the authors of [26] have focused in cases where there is only one emphasized group. As we show in our experiments, choosing the weights achieving sufficient covers for more than one group requires further effort.

Algorithm 1 The MOIM algorithm.

1: **Input:** A network G; emphasized groups $g_1, g_2 \subseteq V$; $k \in [n]$; $t \leq 1-\frac{1}{e}$; an IM algorithm $\mathcal{A}$.
2: **Output:** A k-size seed set S.
3: We run independently the following two procedures:
   i $S_1 \leftarrow$ Run algorithm $\mathcal{A}_{g_2}$, where the seed set size is fixed to $\lceil -\ln(1-t) \cdot k \rceil$.
   ii $S_2 \leftarrow$ Run algorithm $\mathcal{A}_{g_1}$, where the seed set size is fixed to $\lfloor (1+\ln(1-t)) \cdot k \rfloor$.
4: $S \leftarrow S_1 \cup S_2$
5: **if** $|S| < k$ **then**
6: Run $\mathcal{A}_{g_1}$ on the residual network until enough seeds are gathered.
7: **end if**
8: **return** S

The MOIM algorithm is depicted in Algorithm 1. MOIM runs independently two procedures: The first ensures satisfaction of the constraint (line 3.i), while the second maximizes the objective (line 3.ii). We return the union S of the selected seeds (line 4). If S contains less than k seeds, we run $\mathcal{A}_{g_1}$ on the residual problem (by eliminating the respective sets of the seeds selected so far), s.t. additional nodes are added to S (lines 5-7). In practice, this could be achieved by initially running $\mathcal{A}$. Note that this can only improve the accuracy guarantees. In our analysis we assume that the returned set is of size exactly k.

We now state the approximation factor of MOIM.

Theorem 4.1. *For* $0 \leq t \leq 1-\frac{1}{e}$, *MOIM provides a* $(1-\frac{1}{e \cdot (1-t)}, 1)$*-approximation to the Multi-Objective IM problem.*

Example 4.2. Consider again Figure 1, and let k=2. Recall that the optimal solution for g_2 is O_{g_2}=$\{d, f\}$, with $I_{g_2}(O_{g_2})$=2. For t=$1-\frac{1}{e}$, MOIM would be equivalent to running $\mathcal{A}_{g_2}$ with k=2. It would w.h.p. output, if not O_{g_2}, then a set S, s.t. $I_{g_2}(S) \geq 2 \cdot (1-\frac{1}{e}) \approx 1.26$, with no particular regard for g_1 cover, which may be as small as 1.5 (for S=$\{c, f\}$), or as high as 3 (for S=$\{e, f\}$). For t=$1-\frac{1}{\sqrt{e}}$, MOIM runs $\mathcal{A}_{g_1}$ and $\mathcal{A}_{g_2}$ while setting k=1 for both, which would presumably output $\{e\}$ and $\{f\}$ resp., combining for a seed set S s.t $I_{g_1}(S)$=3 and $I_{g_2}(S)$=1.75. This approximated solution comes close to both O_{g_1} and O_{g_2}, in terms of g_1/g_2 cover size, resp.

The time complexity of MOIM depends only on that of its input IM algorithm $\mathcal{A}$, which is assumed to be near optimal [33].

4.2 The RMOIM algorithm

We first describe a theoretical algorithm which, given the optimal cover size of g_2, $I_{g_2}(O_{g_2})$, exactly matches our hardness bound. We then discuss the practical case where $I_{g_2}(O_{g_2})$ is unknown (and can only be approximated in PTIME), proving that the scale of the reduction in the approximation factors is not too high.

Theorem 4.3. *There exists a PTIME randomized algorithm that, given $I_{g_2}(O_{g_2})$, in expectation, outputs a $(1-\frac{1}{e}, 1-\frac{1}{e})$ approximation for the Multi-Objective IM problem.*

We described the reduction from IM to MC suggested in [7], utilized by the RIS framework. We extend this reduction to the multi-objective variants, implying that any algorithm for Multi-objective MC can be extended to Multi-Objective IM, retaining the same guarantees. Therefore, all that is left to prove is that one can get a $(1-\frac{1}{e}, 1-\frac{1}{e})$-approximation for Multi-Objective MC.

Given an instance $\mathcal{I}$ of Multi-Objective MC with m subsets $S_1, ..., S_m$ and two groups $g_1, g_2 \subseteq U$, we construct LP($\mathcal{I}$), the corresponding LP instance, where $Y=|g_2 \setminus g_1|, Z=|g_1 \setminus g_2|, W=|g_1 \cap g_2|$:
variables: $x_1, \ldots, x_m, y_1, \ldots, y_Y, z_1, \ldots, z_Z, w_1, \ldots, w_W$ (x_i is an indicator for selecting S_i, y_i for covering element in $g_2 \setminus g_1$, z_i for covering element in $g_1 \setminus g_2$, and w_i for covering elements in $g_1 \bigcup g_2$)
constraints: $\sum_{i=1}^{m} x_i = k$ (cardinality constraint)

$$\sum_{i:u_j \in S_i} x_i \geq y_j, \sum_{i:u_j \in S_i} x_i \geq z_j, \sum_{i:u_j \in S_i} x_i \geq w_j \text{ (coverage constraint)}$$

$$(\sum_{i=1}^{Y'} y_i \cdot \frac{Y}{Y'} + \sum_{i=1}^{W'} w_i \cdot \frac{W'}{W}) \geq t \cdot I_{g_2}(O_{g_2}) \text{ (size constraint)}$$

$$\forall i \in \{1, ..., m\}, 0 \leq x_i \leq 1; \forall i \in \{1, \ldots, W'\}, 0 \leq w_i \leq 1$$

$$\forall i \in \{1, \ldots, Y'\}, 0 \leq y_i \leq 1; \forall i \in \{1, \ldots, Z'\}, 0 \leq z_i \leq 1$$

objective: maximize $\sum_{i=1}^{Z'} z_i + \sum_{i=1}^{W'} w_i$.
where $I_{g_2}(O_{g_2})$ is the optimal g_2-cover size and Y', Z', W' are the number of sampled nodes from $g_2 \setminus g_1$, $g_1 \setminus g_2$ and $g_1 \cap g_2$, resp.

The solution is determined by the values of the variables x_i, indicating the selected sets. This LP relaxes the Integer LP which precisely models the Multi-Objective MC problem. We can compute an optimal solution by using any LP solver, then apply the following randomized rounding procedure [30]: (1) Interpret the numbers $\frac{x_1}{k}, .., \frac{x_m}{k}$ as probabilities corresponding to $S_1, .., S_m$, resp. (2) Choose k sets independently w.r.t. the probabilities. By adapting the proof in [32], we show that this procedure yields a seed set whose cover, in expectation, for each group separately, is at least a $1-\frac{1}{e}$ fraction of the corresponding optimal cover size, thus proving Theorem 4.3.

Omitting the optimal-value knowledge assumption. As mentioned, the optimal value of the g_2-cover is uncomputable in PTIME. We, therefore, first run a IM_{g_2} algorithm which outputs a seed set S, s.t. $I_{g_2}(O_{g_2}) \cdot (1-\frac{1}{e}) \leq I_{g_2}(S) \leq I_{g_2}(O_{g_2})$. We then set the constraint threshold in $LP(\mathcal{I})$ to $t \cdot (1-\frac{1}{e})^{-1} \cdot I_{g_2}(S)$ instead of $t \cdot I_{g_2}(O_{g_2})$, with the rest of the algorithm remaining the same. This substitution can only increase the constraint threshold, which in turn, reduces the set of valid solutions, possibly diminishing the objective value of the optimal solution subject to the stricter constraint. However, as we prove, the scale of the reduction in α is not arbitrarily large.

The RMOIM algorithm is depicted in Algorithm 2. Given an IM algorithm $\mathcal{A}$, we first run $\mathcal{A}_{g_2}$ to estimate $I_{g_2}(O_{g_2})$ (line 3). Next, using $\mathcal{A}$, we sample the *RR* sets needed for constructing the Multi-Objective MC instance, and build the corresponding LP (lines 4 – 5). Then, we employ an LP solver, obtaining the fractional solution (line 6). Last, we employ the rounding procedure to select k sets for the Multi-Objective MC instance, and return their corresponding nodes as the selected seed-set S (lines 7 – 8).

Given an IM_g algorithm, let S denote its output. We define $\lambda \in [0, \frac{1}{e-1}]$ s.t. $I_g(S) = (1+\lambda) \cdot (1-\frac{1}{e}) \cdot I_g(O_g)$.

Algorithm 2 The RMOIM algorithm.

1: **Input:** A network G; emphasized groups $g_1, g_2 \subseteq V$; $k \in [n]$; $t \leq 1-\frac{1}{e}$; an RIS-based IM algorithm $\mathcal{A}$ and an LP solver.
2: **Output:** A k-size seed set S.
3: $I_{g_2}(\tilde{O_{g_2}}) \leftarrow$ Run $\mathcal{A}_{g_2}$ on the input.
4: RR $\leftarrow$ Construct the *RR* sets using $\mathcal{A}$.
5: LP($\mathcal{I}$) $\leftarrow$ Construct the LP from RR, replacing $t \cdot I_{g_2}(O_{g_2})$ with $t \cdot (1-\frac{1}{e})^{-1} \cdot I_{g_2}(\tilde{O}_{g_2})$.
6: $\vec{X} \leftarrow$ Solve LP($\mathcal{I}$), and output the values for the x_i variables.
7: $S \leftarrow$ Run the randomized rounding procedure on $\vec{X}$.
8: **return** S

Theorem 4.4. *The RMOIM algorithm provides, in expectation, a $((1-\frac{1}{e}) \cdot (1-t \cdot (1+\lambda)), (1+\lambda) \cdot (1-\frac{1}{e}))$ approximation to Multi-Objective IM, where $\lambda \in [0, \frac{1}{e-1}]$.*

The time complexity of RMOIM is dominated by its input LP solver, whose complexity is polynomial in the input size [22].

5 EXTENSIONS

We present an extension of our results to multiple groups, then briefly discuss on alternative problem definitions. We conclude with a discussion regarding a well-studied related problem.

5.1 Multiple Emphasized Groups

The Multi-Objective IM problem naturally extends to multiple groups. Given m emphasized groups, the user can impose size constraints on all but one groups, and subject to these constraints, maximize the cover size of the remaining group. W.l.o.g. let us assume that the user imposed size constraints on the last $m-1$ groups. Given the $m-1$ constraint threshold parameters $t_2, ..., t_m$, analogously to the binary scenario, we can show that a k-size seed set satisfying all constraints can always be found in PTIME if $0 \leq \sum_i t_i \leq (1-\frac{1}{e})$. We prove that in PTIME, one cannot attain an approximation factor dominant over $(1-\frac{1}{e}, \ldots, 1-\frac{1}{e})$. Moreover, our generalized random algorithm matches our lower bound for multiple groups.

Both our algorithms can be generalized to solve the multiple groups scenario. In MOIM we run (independently) $m-1$ IM_{g_i}, $i \in [2, m]$ algorithms, where the seed set size in each algorithm is fixed to $\lceil -\ln(1-t_i) \cdot k \rceil$, and run an $\mathcal{IM}_{g_1}$ algorithm, where the seed set size is fixed to $\lfloor (1+\ln(1-\sum_i t_i)) \cdot k \rfloor$. As in Algorithm 1, we then return the union of the selected seeds. We can show that this algorithm provides a $(1-\frac{1}{e \cdot (1-\sum_i t_i)}, 1, \ldots, 1)$-approximation to Multi-Objective IM with m emphasized groups.

In RMOIM, we first estimate the $I_{g_i}(O_{g_i})$ values for the constrained $m-1$ groups, to include these values in the LP described in Section 4.2. Given an IM_{g_i} algorithm, let S_i denote its output. Recall that $\lambda_i \in [0, \frac{1}{e-1}]$ was defined s.t. $I_{g_i}(S_i) = (1+\lambda_i) \cdot (1-\frac{1}{e}) \cdot I_{g_i}(O_{g_i})$. We prove that RMOIM provides, in expectation, a $((1-\frac{1}{e}) \cdot (1-\sum_i t_i \cdot (1+\sum_i \lambda_i)), (1+\lambda_1) \cdot (1-\frac{1}{e}), \ldots, (1+\lambda_{m-1}) \cdot (1-\frac{1}{e}))$-approximation to Multi-Objective IM with m emphasized groups.

5.2 Alternative problem definitions

We next briefly discuss alternative problem definitions. An alternative variant of Multi-Objective IM is where the user specifies an explicit value constraint (rather than specifying a fraction of the optimal possible value). For instance, continuing with Example 1.2, one may request to maximize the cover over engineers, subject to a constraint requiring that at least 1K researchers

are influenced. Both our algorithms support this variant as well. Specifically, in MOIM, we can run an IM_{g_2} algorithm until it exceeds the constraint value, and with the remaining seeds we run an IM_{g_1} algorithm, which can only improve the guarantees as we no longer overestimate the constraint. In RMOIM, the problem becomes much simpler, since now the exact value for the size constraint is known. Therefore, here RMOIM is optimal as it matches our lower bound (which holds here as well). We focus on the implicit size constraint variant, as the analysis of the explicit value constraint variant is contained in it as a simpler case.

Our definition provides cardinality guarantees over the emphasized groups. An alternative definition may be to constrain the *ratio* of different cover cardinalities. We note that this definition is essentially different form our definition, as maximizing the ratio between the cover cardinalities can dramatically reduce the number of covered users from each group. Therefore, such definition is ill-suited to our motivation where the underlying goal is to reach as many as possible users from the emphasized groups. We further note that the analysis of such ratio-based definitions differs from the one we have provided, and therefore we leave the study of ratio-based constraints for future research.

In our analysis so far the user imposes constraints on all but one group. Our results also support the case where the user imposes constraints on all emphasized groups (see details in [3]).

5.3 Connection to the RSOS problem

The closely related problem of multi-objective maximization of monotone submodular functions subject to a cardinality constraint (known as the RSOS problem) was introduced in [24].

Given m monotone submodular functions $f_i(\cdot), i \in \{1, .., m\}$ and a target value V_i for each function f_i, the goal in the RSOS problem is to find a k-size set A s.t. $\forall i : f_i(A) \geq V_i$, or provide a certificate that there is no feasible solution. A solution S is an $alpha$-approximation if $\forall i : f_i(S) \geq \alpha \cdot V_i$.

In contrast to Multi-Objective IM, where users can specify for each group the *fraction* of the optimal influence that they wish to retain, in RSOS only explicit values can be used. Nonetheless, we establish the connection between the two problems. Specifically, we prove that the two problems are equally hard, and that any algorithm solving RSOS, could in principle also solve Multi-Objective IM. However, as we show in our experiments, top performing RSOS algorithms can only process small networks.

We next briefly present our main results.We restrict our analysis of the RSOS problem to its applicability in an IM setting, s.t. all functions are IM-functions. To simply the presentation, we focus here on the two groups scenario, and defer the analogous results regarding multiple groups to [3].

We reduce RSOS to Multi-Objective IM, showing that any (α, α)-approximation to Multi-Objective IM implies an α-approximation to RSOS. It follows that leveraging existing techniques in RSOS works yields at best an $(1-\frac{1}{e}, 1-\frac{1}{e})$-approximation for Multi-Objective IM, which is an optimum we have already achieved with RMOIM.

Theorem 5.1. *RSOS $\leq_p$ Multi-Objective IM.*

We further provide a reduction in the other direction, showing that any α-approximation algorithm for RSOS, implies an (α, α)-approximation algorithm for Multi-Objective IM.

Theorem 5.2. *Multi-Objective IM $\leq_p$ RSOS.*

Table 1: Datasets.

Datasets	Dimensions	Profile properties
Facebook	|V|=4K, |E|=168K	Gender, Education type.
DBLP	|V|=80K, |E|=514K	Gender, country, age, h-index.
Pokec	|V|=1M, |E|=14M	Gender, age, region
Weibo-Net	|V|=1.5M, |E|=369M	Gender, city.
YouTube	|V|=1M, |E|=3M	-
LiveJournal	|V|=4.8M, |E|=69M	-

However, to do so, we need to know both the optimal cover size of the constrained group $I_{g_2}(O_{g_2})$ (as in RMOIM), and (additionally) the constrained optimal objective value $I_{g_1}(O^*)$. $I_{g_2}(O_{g_2})$ may be estimated, as done in RMOIM, by running an IM_{g_2} algorithm. Here again, we may overestimate this value by a $(1-\frac{1}{e})$ factor, yielding the same guarantees as RMOIM. To efficiently estimate $I_{g_1}(O^*)$, we can examine only $O(log(n))$ guesses for $I_{g_1}(O^*)$, which increases the time complexity of an RSOS algorithm by an $O(log(n))$ factor.

A state-of-the-art algorithm for RSOS, which achieves the optimal $(1-\frac{1}{e})$-approximation, has been introduced in [36]. As we show in our experiments, this algorithm can only process small networks (even without the $log(n)$ multiplicative overhead).

6 EXPERIMENTAL STUDY

We have implemented our prototype in Python 2.7. We use as the input IM algorithm, for both of our algorithms, IMM[1] [33], a top performing IM algorithm. We solve the LP in RMOIM using Gurobi LP solver [2][2]. We have conducted an experimental study to evaluate (1) The quality of results achieved by our algorithms. We demonstrate the advantages of our algorithms in multiple scenarios over real-life datasets, compared to existing and alternatives approaches; (2) The performance of our algorithms in terms of execution times and scalability.

6.1 Experimental setup

We conducted all experiments on a Linux server with a 2.1GHz CPU and 96GB memory. Next, we describe the examined datasets, the considered emphasized groups, the competing algorithms, and the parameters setup.

Datasets. We have focused on social networks which include user profile properties, to characterize the emphasized groups. We have examined 6 commonly used datasets: Facebook, DBLP, Pokec, Weibo-Net, Twitter and Google+ (extracted from [4, 25]). For space constraints, we omit the results over Twitter and Google+, as they were similar to those obtained over the other 4 datasets (depicted in Table 1). To further examine our algorithms scalability, we considered two additional large-scale datasets: YouTube and LiveJournal [25]. These datasets do not include user properties. To nevertheless examine them in our context, we randomly assigned users to emphasized groups (see details below). Following the conventional method as in [28, 34], we set the weight of each edge (u, v) as $w(u, v) = \frac{1}{d_{in}(v)}$, where $d_{in}(v)$ denotes the in-degree of v. To ensure uniformity, undirected networks were made directed by considering, for each edge, the arcs in both directions (as was done in [5]).

[1] We used the corrected version described in [10].
[2] Our code will be publicly available upon acceptance.

Emphasized groups. The benefit that our approach brings is in particular critical for subpopulations that are typically not covered by standard IM algorithms. To identify such groups, we have run, for each network, a grid search over the extracted profile properties. We have considered all groups that are characterized by a single or a combination of two profile properties. For each such group g, we have examined the expected g-cover size of standard IM algorithms, as well as the expected g-cover size of their IM_g counterparts. We are focusing here only on groups in which the results showed that standard IM algorithms tend to overlook their users, while targeted IM algorithms showed that a different choice of seed-set significantly increase their expected cover size. Interestingly, our experiments indicate that all analyzed datasets include several such groups. For example, female Indian researchers in DBLP and females over the age of 50 in Pokec, are typically neglected by standard IM algorithms. Additional examples are provided in [3]. For YouTube and LiveJornal, we have considered random emphasized groups, defined as follows. Given a number $c \in (0, 1]$ (sampled uniformly at random), every node $v \in V$ is a member of the emphasized group with probability of c. Note that this simple definition allows for overlapping emphasized groups of different cardinalities.

Examined scenarios. We examine the following two scenarios: ***Scenario I***. In this scenario the user wishes to maximize the overall influence (g_1), subject to a constraint requiring that at least a given portion of a group's members (g_2) are influenced (a scenario analogous to that of Example 1.1). We focus on this particular scenario as it allows to compare, in a single setting, algorithms for standard IM (that maximize the overall influence), targeted IM (that maximize the influence solely over the g_2 members), and ours. We present the results while setting g_2 to be a group which is not covered by standard IM algorithms (see full details in [3]). We have also run all experiments while choosing all possible pairs of g_1 and g_2 to be groups that are typically not covered by standard IM algorithms. We report that all experiments show similar trends and therefore we omit from presentation these results. ***Scenario II***. Next we consider multiple-groups, to demonstrate the effect of multiple objectives on performance. We present a scenario where the user provides 5 emphasized groups, specifies constraints on 4 of them, and asks to maximize the influence over the remaining group, subject to these constraints. We have also experimented with other numbers of emphasized groups and report that all results have shown similar trends. In real-life scenarios, the number of emphasized groups is typically small [26, 36] and thus we focus on realistic number ranges ($2 - 10$). Here again we have considered groups that are typically not covered by standard IM algorithm.

Competing algorithms. We consider the following baselines.
Standard IM algorithms. We have examined the results of *IMM* [33] and SSA [28], top preforming RIS-based algorithms, as well as SKIM [13] and Celf++ [17], greedy-based IM algorithms. As all algorithms demonstrated similar trends, we detail here only *IMM*.
(Single objective) Targeted IM algorithms. We examine IMM_g, a variant of *IMM* (based on [26]) which maximizes exclusively the cover of a given emphasized group g. In scenario *II* we have defined the target group to be the union of all emphasized groups.
Weighted IM. An alternative is to assign different weights to users, reflecting their relevance to the objectives. The authors of [26] introduced a weighted RIS sampling method, that maximizes the influence over a targeted group. We examined the results for Weighted *IMM* (*WIMM*), a variant of *IMM* which is based on a weighted RIS sampling method presented in [26]. We apply a (multi-dimensional) binary search to find the optimal weights[3]. We examined the results while substituting the weights of users in the constrained group(s) and the objective group with c_i and $1 - \sum_i c_i$, resp[4]., for varying values of $c_i \in [0, 1]$.
We have also examined a variant of *WIMM* that skips the search and instead uses some default weights given as input. **RSOS algorithms.** We examine the RSOS algorithm of [36] (used to solve Multi-Objective IM). Additionally, the authors of [36] have studied the problem of fair resource allocation in IM, and proposed two fairness concepts: *MaxMin*, which maximizes the minimum fraction of users within each group that are influenced, and Diversity Constraints (*DC*), which guarantees that every group receives influence proportional to what it could have generated on its own, based on a number of seeds proportional to its size. They have shown that both fairness concepts can be reduced to RSOS, for which they provided the state-of-the-art algorithm. For completeness, we have included the *MAXMIN* and *DC* baselines. As we show, all RSOS-based algorithms can only process small networks. A more recent fairness-aware IM framework was presented in [15]. However, in this work as well, only small-size networks were examined[5].

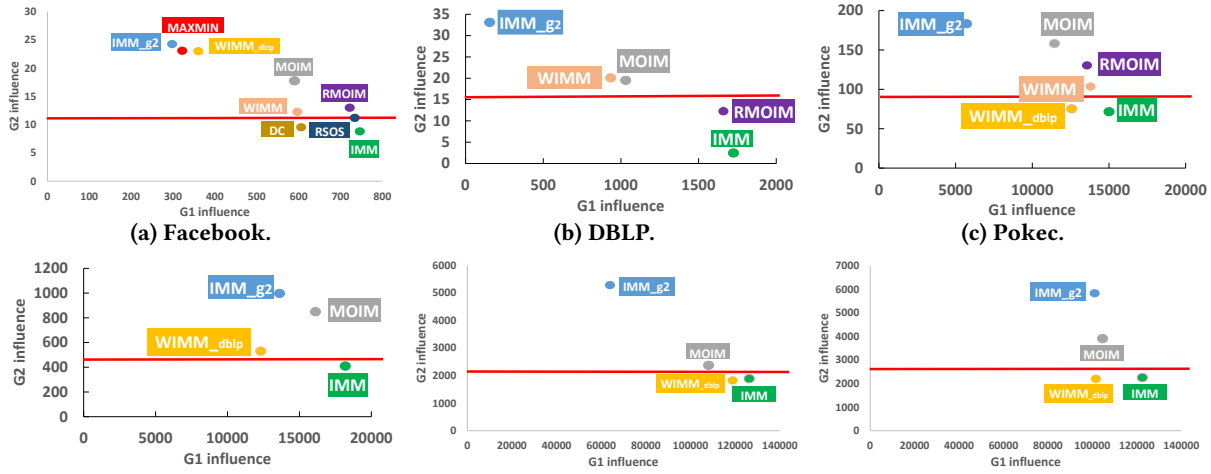
Parameter Settings. Recall that RMOIM requires to estimate $I_{g_i}(O_{g_i})$, the optimal cover cardinality for all constrained groups g_i. For that we use the following estimation strategy (as described in Section 4.2): for each emphasized group g we ran IMM_g for 10 times, selecting the minimal obtained value to derive an estimate for $I_g(O_g)$. Unless mentioned otherwise, we set $k = 20$, and $\epsilon = 0.1$. In scenario *I* we have set the threshold parameter $t = 0.5 \cdot (1 - \frac{1}{e})$, and in scenario *II* we have set the threshold parameters $t_i = 0.25 \cdot (1 - \frac{1}{e}), \forall i \in 1, ..., 4$. We also use, as a default setting, the LT model (when setting uniformly random threshold for every node). In all experiments, the time-out limit is 24 hours (or out of memory exception). For the RSOS baselines, we use the default parameters as provided in [1]. We report for each baseline the averaged measurements of 10 runs.

6.2 Quality Evaluation

Scenario I results. The results are depicted in Figure 2, where the x and y axes represent, resp., the g_1 and g_2 influences, and red lines are the estimated constraint thresholds. A desirable solution should be above (or near) the red lines (i.e., satisfying the constraint), and, at the same time, the right as much as possible (i.e., covering as many g_1 users as possible). For *WIMM*, we present the results obtained by selecting the optimal weights for each dataset (pink points). We have also examined multiple settings of default weights for *WIMM*, however, none of these options yielded satisfying results across all datasets. In particular, the optimal weights per network were different, and to illustrate that, we show how the optimal weights for DBLP operate on the other datasets (yellow points).

In all cases, MOIM managed to match (and sometimes even exceed) the results of *WIMM*, which uses the optimal weights for each dataset. For example, over Facebook, while *WIMM* and MOIM influenced almost the same number of g_1 users (601 and

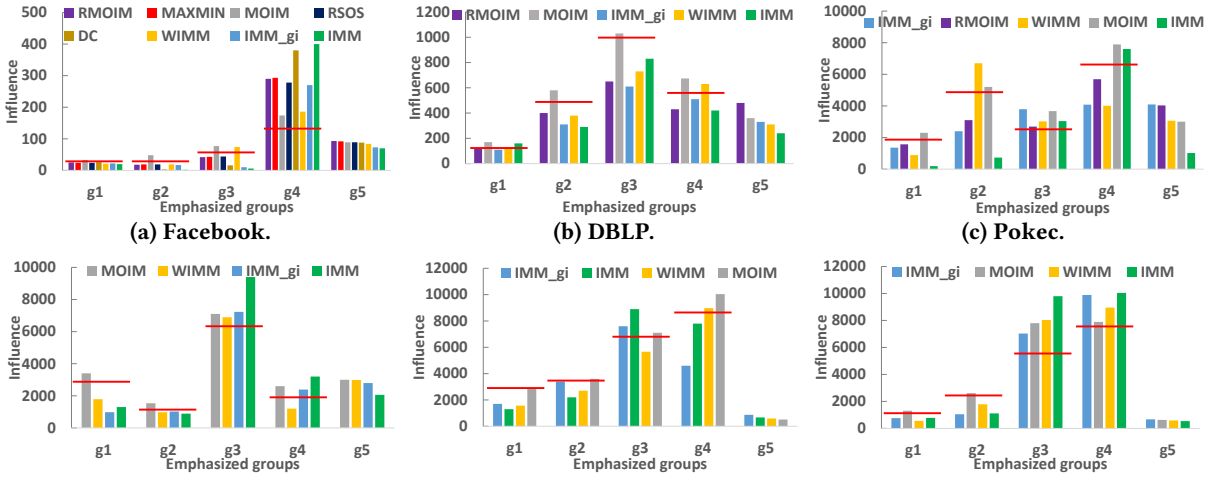
[3]The optimal choice is the one that satisfies all constraints, while maximizing the value for the objective.
[4]Users belong to multiple groups are assigned with the sum of weights of their groups.
[5]In both [36] and [15], the largest examined network included 500 nodes.



(a) Facebook. (b) DBLP. (c) Pokec.

(d) Weibo-Net. (e) YouTube. (F) LiveJournal.

Figure 2: Expected influence with 2 emphasized groups. The red horizontal lines represent the estimated constraints.



(a) Facebook. (b) DBLP. (c) Pokec.

(d) Weibo-Net. (e) YouTube. (F) LiveJournal.

Figure 3: Expected influence with 5 emphasized groups. The red horizontal lines represent the estimated constraints.

599, resp.), MOIM succeeded in covering more g_2 users (19 vs. 12 for MOIM and *WIMM*, resp.). Observe that using the optimal weights for DBLP over Pokec for *WIMM*, result in not satisfying the constraint. The exploration of *WIMM* for optimal weights significantly increases its runtime, making it impractical for massive networks like Weibo-Net, YouTube and LiveJournal (exceeded our time cutoff). In all cases, not only did MOIM satisfy the constraint, it also came very close to the results of IMM_{g_2} in terms of covering g_2 users, which returns the optimal solution. For example, over Pokec, where IMM_{g_2} covered 189 g_2 users, MOIM covers 159, as opposed to *IMM* covering only 73 such users.

Although RMOIM allows for some relaxation of the constraint, it in-fact fully satisfied it in most cases. Moreover, its overall influence was consistently higher than those of *WIMM* and MOIM. In particular, in all but one of the cases, the g_1 influence of RMOIM was very close to that of *IMM*. For example, over DBLP, RMOIM and *IMM* covered 1,661 and 1,712 users, resp., with RMOIM covering over 6 times more g_2 members. RMOIM is incapable of processing massive networks like Weibo-Net (out of memory).

Not surprisingly, the results RMOIM and *RSOS* were similar. Nonetheless, as opposed to RMOIM, all RSOS-based baselines were incapable of even processing medium-size networks (exceeded our time cutoff). Recall that *MAXMIN* aims to maximize the minimum influence over the emphasized groups, and therefore here it behaves similarly to IMM_{g_2} (as $g_2 \subseteq g_1$). As for *DC*, since it guarantees that every group receives influence proportional to what it could have generated on its own, it ignores the constraint. This demonstrates that *MAXMIN* and *DC* are ill-suited for Multi-Objective IM.

Observe that the single objective algorithms were either far from satisfying the constraint (*IMM*) or covered significantly less g_1 users (IMM_{g_2}). Contrarily, both our algorithms succeeded in covering almost as many g_1 users as *IMM*, and almost as many g_2 users as IMM_{g_2}. For example, over DBLP, *IMM* covered only 2 g_2 users and 1,712 users in total (g_1 users), whereas IMM_{g_2} covered 33 g_2 users, and less than 155 in total. MOIM and RMOIM covered 20 and 13 g_2 users, resp., and covered each more than 1,050 users in total. This demonstrates the advantage of our approach over solutions which are focused only on a single objective.

Last, consider Figures 2 (e) and (f). Among all competitors that satisfy the constraints, MOIM has influenced the largest number of users. Interestingly, even though the emphasized groups were randomly generated, *IMM* did not satisfy the constraints. As for IMM_{g_i}, it influences significantly less users than MOIM. This demonstrates that existing single-objective IM algorithms do not ensure the desired balance between the objectives. Note that here the differences in the cover cardinalities among all competitors

were smaller than in other networks. This stems from the fact that the benefit our approach provides is particularly critical for groups that are typically not covered by standard IM algorithms (which is mostly not the case in random emphasized groups).

Scenario II results. The results are depicted in Figure 3, where the y-axis is the influence over the emphasized groups, and red lines represent the estimated constraint thresholds. A desirable solution should be above (or near) the red lines for the constrained groups $g_1, \ldots, g_4$ groups, and, at the same time, should be as high as possible for g_5 (i.e., maximizing the objective). For the *WIMM* baseline we only present the results obtained by using default weights set to 0.2 for all 5 groups (we report that similar results were obtained when using other weighting schemes), as the search for the optimal weights was infeasible in all cases (it exceeded our time cutoff).

MOIM is the only algorithm satisfying all constraints over each dataset. On top of that, its g_5 influence (i.e., objective value) competes nicely with all competitors. For example, over Weibo-Net, MOIM succeeded to cover the greatest number of g_5 members, while over YouTube it covered 510 g_5 members, compared with the best competitor (here - IMM_{g_i}) that covered 810 g_5 users (yet did not satisfy the constraints). In the datasets which RMOIM has managed to process, its g_5 influence was the best or slightly below the best value achieved. E.g., over Pokec, RMOIM and IMM_{g_i} covered 4036 and 4090 g_5 users, resp., while over Facebook and DBLP RMOIM covered the greatest number of g_5 users.

Here again, all RSOS baselines could only process the small Facebook network (exceeded our time cutoff in other datasets), and, as expected, the results of *RSOS* and RMOIM were similar. Here, *MAXMIN* also behaves similarly to RMOIM, however, as noted above, in other scenarios it may behave differently. This stems from the fact that *MAXMIN* optimizes for equality of outcomes, which may be undesirable when some groups are much better connected than others. For instance, if one group is poorly connected, *MAXMIN* would require that a large number of seeds is "spent" on reaching it, even though these seeds may have a relatively small impact on other groups. As the *DC* baseline ignores the constraints, it did not satisfy them.

As opposed to the binary scenario where the objective was to maximize the overall influence, here *IMM* has no advantage over the competitors. Indeed, in all except one of the examined cases, *IMM*'s objective value was the lowest among all algorithms. Furthermore, regarding IMM_{g_i}, as can be seen, covering a greater number of users from one group may come at the cost of significantly reducing the cover sizes of users from other groups. For example, in LiveJournal (Figure 3 (F)), while the g_4 and g_5 cover sizes of IMM_{g_i} were the largest, its g_1 and g_2 cover sizes were significantly lower than the competitors (and below the required constraints). This demonstrates that existing (single-objective) IM algorithms do not ensure the desired balance between the objectives.

6.3 Parameter Tuning

Next, we examine how varying the input parameters affects the results. To illustrate, we present here the results using a range of values for k and t over the DBLP dataset (the other datasets show similar trends). We note that a desirable behavior of a Multi-Objective IM algorithm is as follows. As k increases, we expect both the g_1 (i.e., overall) and the g_2 (i.e., emphasized group) influences to increase as well. As t increases, i.e., the constraint threshold is elevated, the g_2 influence should increase, possibly at the cost of reducing the g_1 (i.e., overall) influence. Naturally, as only our algorithms and *WIMM* take into account the parameter t, other competitors are indifferent to it.

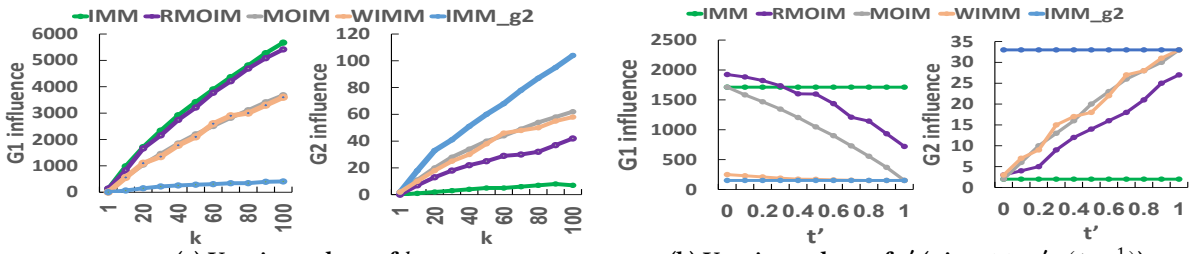
The results are depicted in Figure 4. We first examine Figure 4(a). Interestingly, for all examined k values, the targeted IM algorithm, IMM_g, has shown almost no growth in the overall number of influenced users (less than 400), compared to *IMM* and RMOIM, which, already for $k = 10$, are influencing twice as many users (more than 800). Analogously, for all k values, there is almost no increase in the number of emphasized users influenced by *IMM* (8 such users at most), while IMM_g, already for $k = 10$, influenced twice as many emphasized users (more than 18 such users). Contrarily, MOIM, RMOIM and *WIMM* have demonstrated the desired behavior when k increases. As expected, MOIM, RMOIM and *WIMM*, as t increases, cover a greater number of g_2 users, and fewer users in total, as illustrated in Figure 4(b). Note that in these experiments *WIMM* exhibit the desired behavior, almost identical to that of MOIM. However, as we will see next, its execution times are significantly longer.

6.4 Performance Evaluation

We next measure the cost of enriching the IM problem by incorporating multiple objectives, studying how different parameters affect running times of our algorithms. For brevity, we present the results only for scenario *II*, as the results for scenario *I* show similar trends (see [3]).

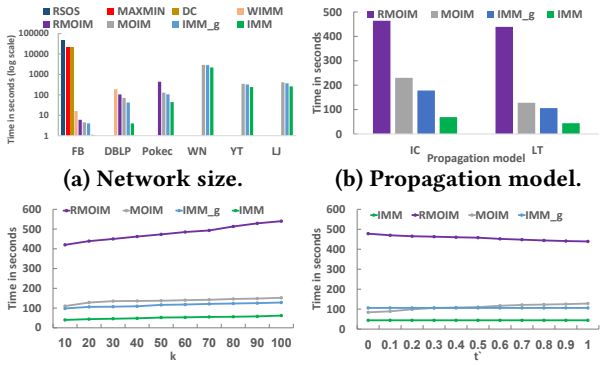
Recall that MOIM runs targeted IM algorithms (i.e., IMM_g) as subroutines. As we show, the overhead for MOIM turns out to be negligible compared to IMM_g, and it can process massive networks efficiently. Naturally, MOIM behaves similarly to its current input algorithm *IMM*, whose optimizations and shortcomings both carry over to MOIM. In particular, as mentioned in [33], when k decreases, so does the optimal expected influence, $I(O)$ (resp. $I_g(O_g)$), in which case it is more challenging for *IMM* (resp. IMM_g) to estimate $I(O)$ (resp. $I_g(O_g)$). Contrarily, for larger k values, *IMM* (resp. IMM_g) is optimized to reuse *RR* sets produced in earlier stages. Thus, the two main factors affecting *IMM* (resp. IMM_g) are k and $I(O)$ (resp. $I_g(O_g)$). Consequently, these factors have a similar effect on MOIM. Regarding RMOIM, we show that solving an LP is indeed costlier than employing an IM algorithm. We will see that when it comes to medium or large scale networks, RMOIM's overhead turns out to be moderate, but when it comes to massive networks it is incapable of processing them. We further show that RMOIM's scalability is not affected by the same factors as MOIM, and its running times are barely affected by those of its input IM algorithm.

Network size. . We first report the running times for the cases presented above in Figure 5(a). Naturally, all competitors' running times increase for larger networks. Although we see that MOIM and RMOIM are naturally slower than *IMM* and IMM_g, they run in approximately 2 and 7 minutes, resp., even on Pokec, which includes 1M nodes and 14M edges. That is, both our algorithms can process large-scale networks in feasible running times. Importantly, note that the running times of MOIM are very close to those of IMM_{g_i} (i.e., MOIM and IMM_{g_i} have processed YouTube in 5.7 and 5.3 minutes, resp.). When it comes to massive networks such as Weibo-Net, while MOIM processed it in less than 49 minutes (in comparison, IMM_{g_i} processed it in 47 minutes), RMOIM can not process it, since the LP program was too big for the LP solver to handle (out of memory). According to our experiments, RMOIM is feasible for graphs including up



(a) Varying values of k. (b) Varying values of t' (t is set to $t' \cdot (1 - \frac{1}{e})$).

Figure 4: The expected influence of different baselines on the DBLP network, using varying values of k and t.



(a) Network size. (b) Propagation model.

(c) Seed-set size k. (d) Constraint threshold.

Figure 5: Averaged execution times (for scenario *II*).

to 20M edges and nodes. Regarding *WIMM*, as it searches for optimal weights, its running times were significantly longer than both our algorithms. For example, on Facebook, it took *WIMM* 16 seconds - almost 4 times slower than MOIM (which ended after 4.5 seconds). Observe that all RSOS-based algorithms ran in more than 6 hours, even on the small Facebook instance network.

In what follows we focus on the Pokec dataset, as this is the largest dataset RMOIM can process. We omit the results of the RSOS-based and *WIMM* baselines, as they cannot process it.

Propagation model. We present the effect of the propagation model on running times in Figure 5(b). As reported in [5], while *IMM* scales well under the LT model, it shows inferior performance under the IC model, as it samples more *RR* sets. Consequently, all *IMM* variants, MOIM included, run slower under the IC model. Indeed, it took all *IMM* variants almost twice the time to process Pokec when using the IC model. Contrarily, as RMOIM is less sensitive to the increase in the number of *RR* sets, and it behaves similarly under both propagation models (the difference was less than a minute). As explained in [5], besides *IMM*, multiple top performing IM algorithms are not robust across different propagation models (e.g., [18], [34]). This property of *IMM* is naturally carried over to MOIM. In cases where the user is interested in a different propagation model, she can take a different IM algorithm optimized for this model (e.g., [13] for IC) as an input for MOIM.

Seed-set size. . In Figure 5(c) we examine the effect of the parameter k on running times. As mentioned, when k increases *IMM* employs an optimized computation and hence we observe almost no change in running times for all *IMM* variants, MOIM included. This behavior of MOIM is a consequence of employing *IMM*, and therefore using an alternative IM algorithm (e.g., [17]) could lead to a linear growth in running times. As expected, RMOIM demonstrates nearly linear growth as a function of k, as more k-size seed sets are considered.

Constraint threshold parameter. . In Figure 5(d) we examine how the parameters $t_i, i \in [1, 4]$ affect performance. Here we tested all t_i values of the form $t_i = 0.25 \cdot t' \cdot (1 - \frac{1}{e})$, where $t' \in [0.1, 0.2, \ldots, 1]$. Note that this parameter only affects the behavior of our algorithms. In MOIM it dictates the required seed-set size for the procedures it employs. Observe that when all $t_i = 0$ MOIM only runs IMM_{g_5}, while for other t_i values it employs 5 versions of IMM_{g_i} with smaller k values, therefore it cannot use *IMM* optimizations for large k values. On the other hand, as the solution space becomes smaller for higher t_i values (i.e., less k-size seed-sets satisfy the constraint), the running time of RMOIM decreases.

7 RELATED WORK

The seminal work of [23], the first to formulate the IM problem, has motivated extensive research [5, 13], which can be classified into three main approaches: (i) The greedy framework [18, 23, 29], which iteratively adds nodes to the seed-set, maximizing the expected marginal influence gain; (ii) The RIS framework [7], where, while retaining optimal accuracy, running times were gradually improved, resulting in highly scalable algorithms [20, 28, 33]; (iii) In cases where scalability is preferred over accuracy, there are heuristic algorithms that have been shown to perform well in practice (e.g., [11]), despite not having theoretical guarantees. Any greedy or RIS-based IM algorithm can be embedded in MOIM, retaining the same features and drawbacks. In our experiments we have examined the results of top performing IM algorithms (e.g., [17, 33]), showing them all to be ill-suited for the Multi-Objective IM problem.

An extension of IM, which we also examined in our experiments, is *targeted IM*, where the goal is to maximize the influence over a target group of users [6, 9, 26]. As demonstrated, this extension as well is ill-suited for the Multi-Objective IM problem, as maximizing the influence over one group of users may come at the cost of influence decrease for other groups. Therefore, unlike our solutions, it does not provide theoretical guarantees for the influence over each emphasized group separately.

Multi-Objective optimization problems (also known as Pareto optimization) involve several (possibly conflicting) objectives, which are required to be optimized simultaneously. Such problems have been studied in numerous fields, including economics [27], finance [35], social-network analysis [19] and engineering [14]. A classic approach to tackle such problems, which was adopted by targeted IM algorithms [26, 31]), is the weighted-sum method (e.g., [21]), which scalarizes the objectives into a single objective, by assigning to each objective a user-defined weight (which is chosen in proportion to its relative importance). In the IM setting, the relative weights of users in the overall influence sum are altered in accordance with a context-based function

[6, 9, 26]. The main disadvantage of this method is the difficultly in setting the weights obtaining the desired trade-off between the objectives. Indeed, as we show in our experiments, adopting the weighted-sum approach for our context requires an exploration for the optimal weights which strike the desired balance. Hence, this solution results in poor performance.

An alternative, more direct approach to multi-objective optimization problems is the *constraints method* (e.g., [12]), that transforms all except one objectives into constraints, optimizing the remaining objective subject to these constraints. A typical challenge when applying this method is that the constraints have to be chosen within the minimum/maximum values of the individual objectives (which are generally unknown). Our solution follows this approach, which enables the user to prioritize her objectives and provides lower bound guarantees for all of them. As mentioned, to assist the user in choosing the minimum values of the objectives, IM-Balanced indicates to the user the range of possible constraints per objective.

We have discussed on the connection between Multi-Objective IM and the RSOS problem [24]. The authors of [8] provided an optimal $(1-\frac{1}{e})$-approximation algorithm for RSOS (assuming that number of objectives is $m = \Omega(k)$), which runs in $O(n^8)$. Udwani [37] has recently introduced two more efficient algorithms. The first is an optimal $(1-\frac{1}{e})$-approximation algorithm, which runs in $\tilde{O}(mn^8)$. The second is a more efficient algorithm which runs in $O(n \log m \log n)$, yet achieves only a $(1-\frac{1}{e})^2$ approximation. More recently, the authors of [36] remedy this gap by providing an optimal $(1-\frac{1}{e})$-approximation algorithm, whose runtime is comparable to the second algorithm of Udwani. As mentioned, we have included this algorithm in the experimental study, showing that, unlike our algorithms, it fails to process large networks.

8 CONCLUSION AND FUTURE WORK

We have presented the IM-Balanced system, which employs Multi-Objective IM, a refined notion of the IM problem, handling multiple objectives. We motivate the practical relevance of this problem, and propose two algorithms: MOIM and RMOIM. IM-Balanced employs RMOIM for social networks including up to 20M users and links, and MOIM for larger networks. Our experimental study demonstrates the advantages of our algorithms in multiple real-life scenarios, compared to alternative approaches.

We are currently pursuing complementary Multi-Objective IM definitions, e.g., definitions aiming to maximize the *ratio* of different cover cardinalities, inspired by recent work on fairness-aware IM [15, 36]. We identify several interesting directions for future research, which include confirming the tightness of MOIM, and identifying other optimum values for Multi-Objective IM.

Acknowledgment. This work has been partially funded by the Israel Science Foundation, the Binational US-Israel Science Foundation, Tel Aviv University Data Science center, and eBay Israel.

REFERENCES

[1] 2019. Code for the paper: Group-Fairness in Influence Maximization. https://github.com/bwilder0/fair_influmax_code_release.
[2] 2020. Gurobi LP solver. http://www.gurobi.com/.
[3] 2020. Technical Report. https://bit.ly/2V9nXIS.
[4] AMiner 2018. AMiner datasets. https://aminer.org/data-sna.
[5] Akhil Arora, Sainyam Galhotra, and Sayan Ranu. 2017. Debunking the Myths of Influence Maximization: An In-Depth Benchmarking Study. In *SIGMOD*.
[6] Cigdem Aslay, Nicola Barbieri, Francesco Bonchi, and Ricardo A Baeza-Yates. 2014. Online Topic-aware Influence Maximization Queries.. In *EDBT*.
[7] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. 2014. Maximizing Social Influence in Nearly Optimal Time. In *SODA*. Society for Industrial and Applied Mathematics.
[8] Chandra Chekuri, Jan Vondrak, and Rico Zenklusen. 2010. Dependent randomized rounding via exchange properties of combinatorial structures. In *FOCS*. IEEE.
[9] Shuo Chen, Ju Fan, Guoliang Li, Jianhua Feng, Kian-lee Tan, and Jinhui Tang. 2015. Online Topic-aware Influence Maximization. *PVLDB* (2015).
[10] Wei Chen. 2018. An issue in the martingale analysis of the influence maximization algorithm IMM. In *International Conference on Computational Social Networks*. Springer.
[11] Wei Chen, Yajun Wang, and Siyu Yang. 2009. Efficient Influence Maximization in Social Networks. In *SIGKDD*.
[12] Kenneth Chircop and David Zammit-Mangion. 2013. On-constraint based methods for the generation of Pareto frontiers. *Journal of Mechanics Engineering and Automation* (2013).
[13] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F. Werneck. 2014. Sketch-based Influence Maximization and Computation: Scaling Up with Guarantees. In *CIKM*. ACM.
[14] Kalyanmoy Deb and Rituparna Datta. 2012. Hybrid evolutionary multi-objective optimization and analysis of machining operations. *Engineering Optimization* (2012).
[15] Golnoosh Farnad, Behrouz Babaki, and Michel Gendreau. 2020. A Unifying Framework for Fairness-Aware Influence Maximization. In *WWW*.
[16] Sahy Gershtein, Tova Milo, Brit Youngmann, and Gal Zeevi. 2018. IM Balanced: Influence Maximization Under Balance Constraints. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. ACM.
[17] Amit Goyal, Wei Lu, and Laks VS Lakshmanan. 2011. Celf++: optimizing the greedy algorithm for influence maximization in social networks. In *Proceedings of the 20th international conference companion on World wide web*. ACM.
[18] Amit Goyal, Wei Lu, and Laks V.S. Lakshmanan. 2011. CELF++: Optimizing the Greedy Algorithm for Influence Maximization in Social Networks. In *WWW*. ACM.
[19] R Chulaka Gunasekara, Kishan Mehrotra, and Chilukuri K Mohan. 2014. Multi-objective optimization to identify key players in social networks. In *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2014)*. IEEE.
[20] Keke Huang, Sibo Wang, Glenn Bevilacqua, Xiaokui Xiao, and Laks V. S. Lakshmanan. 2017. Revisiting the Stop-and-stare Algorithms for Influence Maximization. *PVLDB* (2017).
[21] C-L Hwang and Abu Syed Md Masud. 2012. *Multiple objective decision making—methods and applications: a state-of-the-art survey*. Vol. 164. Springer Science & Business Media.
[22] Narendra Karmarkar. 1984. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. ACM.
[23] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the Spread of Influence Through a Social Network. In *KDD*.
[24] Andreas Krause, H Brendan McMahan, Carlos Guestrin, and Anupam Gupta. 2008. Robust submodular observation selection. *Journal of Machine Learning Research* (2008).
[25] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[26] Yuchen Li, Dongxiang Zhang, and Kian-Lee Tan. 2015. Real-time Targeted Influence Maximization for Online Advertisements. *PVLDB* (2015).
[27] SJ Mardle, Sean Pascoe, and Mehrdad Tamiz. 2000. An investigation of genetic algorithms for the optimization of multi-objective fisheries bioeconomic models. *International Transactions in Operational Research* (2000).
[28] Hung T Nguyen, My T Thai, and Thang N Dinh. 2016. Stop-and-stare: Optimal sampling algorithms for viral marketing in billion-scale networks. In *SIGMOD*. ACM.
[29] Naoto Ohsaka, Takuya Akiba, Yuichi Yoshida, and Ken-Ichi Kawarabayashi. 2014. Fast and Accurate Influence Maximization on Large Networks with Pruned Monte-Carlo Simulations. In *AAAI*. AAAI Press.
[30] Prabhakar Raghavan and Clark D Tompson. 1987. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica* (1987).
[31] Chonggang Song, Wynne Hsu, and Mong Li Lee. 2016. Targeted Influence Maximization in Social Networks. In *CIKM*. ACM.
[32] David Steurer. 2014. Max Coverage—Randomized LP Rounding. http://www.cs.cornell.edu/courses/cs4820/2014sp/notes/maxcoverage.pdf.
[33] Youze Tang, Yanchen Shi, and Xiaokui Xiao. 2015. Influence Maximization in Near-Linear Time: A Martingale Approach. In *SIGMOD*.
[34] Youze Tang, Xiaokui Xiao, and Yanchen Shi. 2014. Influence maximization: Near-optimal time complexity meets practical efficiency. In *SIGMOD*. ACM.
[35] Ma Guadalupe Castillo Tapia and Carlos A Coello Coello. 2007. Applications of multi-objective evolutionary algorithms in economics and finance: A survey. In *2007 IEEE Congress on Evolutionary Computation*. IEEE.
[36] Alan Tsang, Bryan Wilder, Eric Rice, Milind Tambe, and Yair Zick. 2019. Group-fairness in influence maximization. *arXiv preprint arXiv:1903.00967* (2019).
[37] Rajan Udwani. 2018. Multi-objective maximization of monotone submodular functions with cardinality constraint. In *NeurIPS*.
[38] Vijay V Vazirani. 2013. *Approximation algorithms*. Springer Science & Business Media.

Subjectivity Aware Conversational Search Services

Yacine Gaci
LIRIS - Université Lyon 1, France
yacine.gaci@univ-lyon1.fr

Jorge Ramírez
University of Trento, Italy
jorge.ramirezmedina@unitn.it

Boualem Benatallah
University of New South Wales, Australia
& LIRIS - Université Lyon 1, France
b.benatallah@unsw.edu.au

Fabio Casati
University of Trento, Italy
& ServiceNow, USA
fabio.casati@gmail.com

Khalid Benabdeslem
LIRIS - Université Lyon 1, France
khalid.benabdeslem@univ-lyon1.fr

ABSTRACT

Online users are becoming increasingly dependent on Web services in choosing among products and services. This recent trend is motivated by the integration of conversational agents which took the human-machine interaction to unprecedented levels of ease, using natural language as a communication medium. Given the success of these systems, users are constantly switching to experiential search, providing utterances that are intrinsically subjective such as looking for a restaurant with a romantic ambiance, creative cooking or nice staff. Current Web services are unfortunately unable to decipher the subjective signals present in user utterances and only support objective attributes that are listed in service descriptions (e.g., restaurant address, cuisine, price range).

To make the most of dialog systems, they must be able to detect subjective attributes in user utterances and filter responses according to user subjective preferences. This paper presents a framework and techniques that augment conversational search services with capabilities to understand and reason about subjective user utterances. We propose novel subjective tag-based indexing of information services. We discuss automatic subjective tag extraction from both user utterances and online reviews using state of the art machine learning techniques such as BERT, adversarial training and data programming. Experiments show that the proposed techniques outperform existing information retrieval systems and the search mechanisms provided by well-known web search services such as Yelp.

1 INTRODUCTION

Digital services and online reviews are widely used in day-to-day decisions [14], such as providing recommendations or opinions regarding which restaurant to eat, which research paper to read or even who to vote for in elections. When we make decisions, recent studies show that we are prone to lean toward subjective data focusing on past experiences rather than relying solely on objective information (e.g., type of food served by a restaurant or an address of a restaurant) [31]. For example, when we set out to choose between two restaurants, we are attracted by the ones offering great experiences such as delicious food, brilliant atmosphere, friendly staff or romantic ambiance in addition to deciding based on factual information such as restaurant locations or specific types of cuisine [39]. Often, we find experiential and subjective information in online reviews because they reflect user opinions and experiences [14]. Techniques from opinion mining and information retrieval (IR) [32] can be used to extract knowledge from reviews. However, such techniques usually lack the necessary precision to obtain meaningful and accurate subjective information due to their keyword-based search nature [31]. On the other hand, online reviews are expressed in natural language which is very nuanced and intricate, thus needing more effective extraction techniques. In the same line of argument, information retrieval systems are heavily manual for the users given that users need to try different combinations of keywords and query styles before having to compare between the results in a manual and labor-intensive way. Hence, decisions made through traditional information retrieval systems are generally sub-optimal [14].

Capturing and reasoning about subjective information has been explored at various levels [31]. Some techniques explored ratings (e.g., star ratings) to represent aggregated user opinions on entities or services [59]. Rating-based techniques do not consider reviews content but they rather provide aggregated numerical or symbolic values which are hard for users to express accurately. For example, a star rating of three out of five might give the impression that the restaurant is average in all aspects but in reality, it may serve delicious food but employs unhelpful waitstaff, which made the reviewer balance out her final rating. Another line of subjectivity-related research translates numerical attribute values to linguistic values (e.g., translation of prices to linguistic values, such as {"cheap", "fair", "costly", "expensive"}) using insights from fuzzy logic [26]. Nonetheless, such methods involve objective attributes, whose values are to be translated into subjective linguistic variations, leaving the space of the inherently subjective attributes untouched.

More recently, [31] proposed techniques extending database systems to account for both objective and subjective attributes and support subjective database queries. Nevertheless, in order to use [31], the database schema, and hence the subjective attributes, must be defined beforehand. Unfortunately, it is not always easy to identify which attributes to include in the schema and what their precise meaning is [14]. Besides, while such extensions [31] augment structured query languages with subjectivity support, they presuppose technical expertise comparable to that of professional database users, who can express (complex) SQL queries. Therefore, there is a need for more advanced techniques to empower all users to benefit from subjectivity-aware services in performing their day-to-day activities in a digitally enabled world.

At the same time, conversational Artificial Intelligence (AI) and its instantiation in the form of messaging or chat bots (also called task-oriented conversational bots) emerged as a new paradigm to naturally access services and perform tasks through natural language (text or voice) conversations with software services and humans. Thousands of bots have already been used in a variety of significant use cases, e.g. tourism, travel, office tasks, healthcare, e-commerce, education, and e-government services. On the downside, the current generation of conversational bots does not handle subjective information users ought to include in their utterances and often ignores them, leading to user dissatisfaction.

In this work, we propose *SACCS* (Subjectivity Aware Conversational Search Service), consisting of a Natural Language Understanding (NLU) framework and techniques, that combine the usefulness of including subjective information in the search utterances and the flexibility of utilizing natural conversations to interact with users. A key feature of *SACCS* is the ability to automatically and dynamically extract subjective information from user utterances and online reviews without explicitly defining them beforehand. Achieving such an objective faces several difficult issues, the most challenging of which is due to the expressiveness and complexity of natural language, i.e. the same subjective information can be expressed using various phrases. For instance, both *"The food is phenomenal"*, *"Very tasty plates of food"* or *"Really good food"* denote the deliciousness of food. To address this issue, we introduce the concept of *subjective tags*. Briefly stated, a subjective tag denotes subjective information in user utterances and online reviews. For example, the review sentence *"This restaurant serves really good food and the service is really quick"*, is tagged with {*delicious food*, *quick service*} subjective tags. The use of tags provides a powerful mechanism to reason about subjective information in user utterances and online reviews (e.g., organization, navigation, summarization, matching and understanding of subjective information). Building upon advances in opinion extraction, in our approach a subjective tag is represented as concatenation of an *aspect* term and an *opinion* term [32]. The aspect term denotes the feature being described and the opinion term characterizes this feature. For example, *delicious food* is a subjective tag wherein *food* is the aspect while *delicious* is the opinion. *SACCS* marks each review with a corresponding set of subjective tags.

In this paper we make the following contributions to overcome challenges related to extracting subjective tags from user utterances and online reviews as well as using them to support subjectivity-aware human-bot natural language conversations:

- We introduce a framework that augments task-oriented dialog systems with subjective filters. Search services are augmented with subjective tag based search and indexing [36]. Each subjective tag in the index is mapped to a list of reviews and entities (e.g. restaurants, books, hotels...).
- We provide a novel subjective tag extraction pipeline that is robust against variations of natural language. Tagging labels each word in a natural language sentence as being either an aspect, an opinion or neither. We train a subjective tag extraction model (called extractor) in an adversarial fashion, wherein the adversary [13, 38] adds *informed* perturbations to natural language sentences. This allows the tag extraction model to learn the possible variations in language and update its parameters accordingly.
- After the aspects and opinions have been extracted, there is a need to pair each aspect with its corresponding opinion in order to construct subjective tags. We propose two novel heuristics for pairing an aspect to an opinion. These heuristics aim to overcome the limitation of word-based distance approaches for pairing an aspect term to an opinion term [31, 56]. The first heuristic relies on the distance between aspects and opinions in the review parse trees [24, 25, 41]. For example, the opinion *professional* would be wrongfully paired with the aspect *decor* in the review *"The staff is friendly, helpful and professional. The decor is beautiful"* when relying on word distance alone. However, when using a parse tree to represent the above review, the two sentences *"The staff is friendly, helpful and professional"* and *"The decor is beautiful"* belong to two separate sub-trees. Consequently, the opinion *professional* will be closer to the aspect *staff* than the aspect *decor* because *professional* and *staff* belong to the same sub-tree. While more effective than traditional word distance techniques, this heuristic has the following limitations: (i) In long sentences, the different aspects and opinions may not be separated into their own sub-trees. In this case, this heuristic provides the same result as word distance methods. (ii) The generated parse tree will be incorrect when there are typos or punctuation errors in the review. The second heuristic is proposed to overcome these limitations. It relies on attention mechanism [3, 17, 54] to distribute the attention of an aspect term among the different opinion terms. These heuristics are combined using a data programming paradigm [2, 49] to : (i) pair opinion and aspect terms in natural language sentences in an unsupervised model, or (ii) automatically generate training data from online reviews to build a supervised pairing model.
- We evaluate the performance of the proposed techniques using crowd sourced data. Experiments show that *SACCS* provides better results than IR systems. Besides, the tagger improves upon state of the art by up to 4.93% in F1 scores while the supervised pairing method adds 3.03 points in accuracy.

This paper is organized as follows: We first discuss related work. We then introduce the architecture of *SACCS* in Section 3. The extraction of subjective tags is discussed in Sections 4 and 5. Section 4 describes the tagging while Section 5 details pairing. Finally, Section 6 discusses the evaluation of the proposed techniques.

2 RELATED WORK

Our work lies at the intersection of two areas: Subjectivity search, and aspect/opinion extraction.

Subjectivity Search. Despite the overwhelming importance of subjective information in the decision making process, relatively little effort focused on understanding and measuring the effect of subjectivity in user decisions [31]. This task has been traditionally delegated to standard information retrieval systems which provide a keyword-based search, and a synonym expansion at best [36, 52]. Systems that incorporate subjective filters in their data models attracted the attention of the research community only recently. Perhaps the closest work to ours is [31] which aimed to augment traditional database systems with subjective attributes. Their approach is different from ours in that their

subjective attributes are part of a database schema itself, which should be explicitly defined by a database designer beforehand. The query interfaces require the users to have precise knowledge about source schemas too. In our approach, subjective tags are dynamically extracted from user utterances as they interact with the system, thus increasing flexibility and productivity.

[27] built a tunable high-precision knowledge base with both factual and subjective attributes. To do so, they predefined a list of attributes (e.g. GOOD_VIEW, KID_FRIENDLY, HAS_HIGH_CHAIRS) and asked crowd workers to assess whether an entity (in their case, they used locations in Google Maps) has each attribute or not. They then modeled user consensus with Beta distributions. The major limitation of this approach is the increasing cost of crowd workers when adding new locations, new attributes or even changing the domain. Besides, crowd-sourced data suffers from data quality problems, mainly due but not limited to the inherent subjectivity in the task at hand. Also, the subjective attributes in [27] are set at design time and not learned from user interactions as we do.

Prior works also tackled the problem of subjectivity and opinions in various domains [29, 37, 59]. Most of them capture a narrow aspect of subjectivity by prompting the reviewers to rate the objects they write about. We often find these in e-commerce services in the form of star ratings which aggregate opinions of all sub-parts of the object and act as a proxy for the overall user satisfaction. This suffers from coarse granularity because the star rating skips the details we might be interested in and only gives one global assessment of the reviewer's true feeling.

Another body of research aims to translate objective facts into subjective phrases [20, 26, 50, 60]. The dominant example is the price which is mapped to a set of subjective phrases such as {"cheap", "fair", "costly", "expensive"} depending on comparisons between the price value and a set of thresholds. This approach only deals with translating objective attributes whose values are indisputable. It leaves the space of the inherently subjective attributes such as food deliciousness or room cleanliness untouched.

Aspect Opinion Extraction. The problem of extracting aspects from review texts is a long standing one in the Natural Language Processing (NLP) literature [32]. However, most previous work focused on identifying the aspects only and measuring their quantitative sentiment polarity (as being positive, negative or somewhere in between). This task is often referred to as *Aspect-Based Sentiment Analysis* (ABSA) [32].

Existing approaches include rule-based, feature-engineering-based and deep-learning-based approaches [56]. In a rule-based approach, to classify the aspect terms as positive or negative, a lexicon is used along with handcrafted sentiment values [18, 19]. Feature-based approaches [22, 30] train a classifier to extract the aspect terms with manually defined features. Both rule-based and feature-based solutions are labor-intensive and highly demanding in terms of effort and time. Deep-learning-based approaches [33, 55, 56], aside from having superior performance than the previous two methods, extend the extraction to opinion terms as well. While [55] used recursive neural networks, [56] employed an attention-based architecture. The motivation behind both approaches is the necessity to link aspects to opinions. [31] employed BERT sentence embeddings [7] with a standard classifier that classifies each word in the sentence into either Aspect, Opinion or Other. In the same spirit, we use BERT as an embedding layer along with a BiLSTM-CRF classification model. We also leverage adversarial training to handle potential variations in the language. Experiments show that *SACCS* 's extractor yields better performance in various test benchmarks.

Table 1: An example of an inverted index with degrees of truth for each subjective tag and restaurant pair

Tag	Restaurants
good food	Vue du Monde (0.89) Anchovy (0.76) Pizza Hut (0.82)
nice staff	Vue du Monde (0.92) Pizza Hut (0.63)
creative cooking	Anchovy (0.94) Pizza Hut (0.34) Kazuki's (0.85)
fast delivery	Anchovy (0.13) Pizza Hut (0.75) McDonald's (0.74)

3 SUBJECTIVE TAG BASED INDEXING AND FILTERING IN CONVERSATIONAL SEARCH SERVICES

To describe the pipeline of *SACCS* , we begin with illustrating how subjective tags are used. We then move on to show how *SACCS* constructs these tags and how it uses them to answer complex and subjective user utterances. It should be noted that, while the proposed techniques are not domain specific, we choose the restaurants domain as a use case in this paper in order to illustrate the components of the proposed pipeline. We also assume that the underlying dialog system is already equipped with intent recognition [15, 23, 46] and slot filling techniques [4, 12]. Briefly stated, intent recognition allows the identification of user intents from user utterances. For instance, from the following user utterance: *"I want to eat Italian food near Lyon in a romantic ambiance"*, the dialog system identifies that the user is searching for a restaurant. Once an intent is identified, the system also extracts what is called slots (e.g., the type of cuisine (Italian), the location of the restaurant (Lyon)). The chatbot then delegates the search intent to a search API that retrieves a list of restaurants filtered by objective criteria. The goal of *SACCS* is to re-filter this list to only keep the restaurants which offer a romantic ambiance.

3.1 Subjective Tag Index

In order to use subjective tags, *SACCS* leverages an inverted index data structure [36]. Table 1 shows a snippet of what the index might look like [1]. Each subjective tag points to a set of entities (in this case restaurants) whose reviews include mentions of the subjective tag. For example, *good food* in Table 1 points to *Vue du Monde*, *Anchovy* and *Pizza Hut*, meaning that the reviews of these restaurants mention the deliciousness of the food cooked there. Also, every entity is accompanied by a degree of truth. Informally, a degree of truth associated to a tag measures the degree of certainty that *SACCS* exhibits when marking an entity with the tag. In the case of Table 1, *Vue du Monde* is more likely to have a nice staff than *Pizza Hut* (a degree of truth of 0.92 compared to 0.63). The degrees of truth are computed automatically by *SACCS* .

[1] The degrees of truth reported in the table are for illustration only and do not reflect the quality of these restaurants in the real world

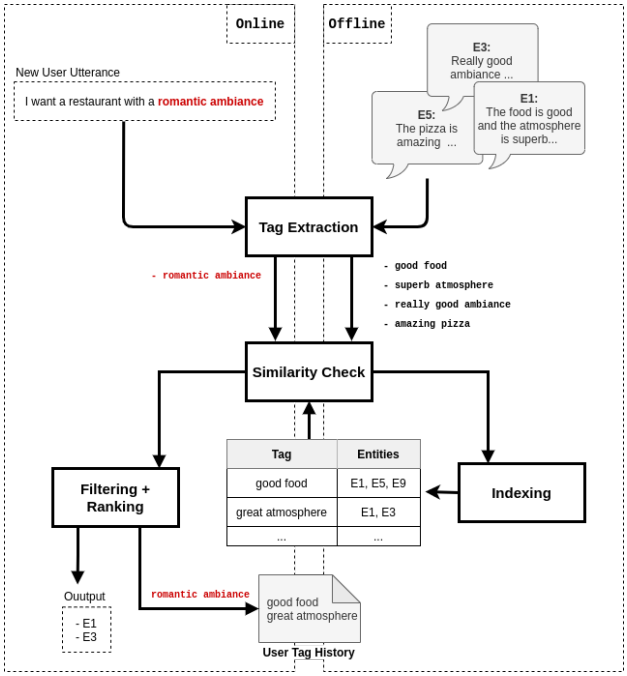


Figure 1: Architecture of *SACCS*

After collecting the set of subjective tags, *SACCS* needs to associate each tag with a set of entities, as depicted in Table 1. Association, or mapping, between a tag and an entity is based on similarity scores [53]: *SACCS* reads online reviews of the entity and extracts all subjective mentions from it. It then proceeds to compute similarities between the subjective tags in the index with those extracted from the reviews. If the similarity exceeds a predefined threshold, *SACCS* includes the corresponding entity to the index. Figure 1 illustrates this process.

The index in Figure 1 contains two subjective tags: *good food* and *great atmosphere*. Suppose we have three entities (*E1, E3* and *E5*) each having only one review. The extractor component extracts subjective tags from the reviews, in this case *good food*, *superb atmosphere*, *really good ambiance*. In the next step, the similarity checker computes similarity scores between the review tags and the index tags. Each time a similarity exceeds a specified threshold, the indexer adds the corresponding entity to the appropriate subjective tag in the index. Following the same example in Figure 1, *E1* and *E5* are both included as mappings to the subjective tag *good food* because their reviews both mention it (*good food* and *amazing pizza* for *E1* and *E5* respectively). However, the review of *E3* only mentions the ambiance; hence *SACCS* does not add it as a mapping to *good food*. We use conceptual similarity which, in addition to the individual meaning of words, also considers their nature or concept, for example pizza being a type of food [2]. Conceptual similarity has been shown to work better on short phrases such as subjective tags than cosine similarity. When building the index, *SACCS* automatically computes the degrees of truth of an entity e with respect to *tag*. The exact formula is shown in Equation 1.

[2]Conceptual similarity is outside the scope of this paper and may be subject to another submission

$$Deg\_truth(tag, e) = \frac{log(|R_e| + 1)}{|T_e^{tag}|} * \sum_{t \in T_e^{tag}} Sim(tag, t) \quad (1)$$

Where R_e is the set of entity e's reviews and T_e^{tag} is the set of subjective tags automatically extracted from R_e and whose similarity score exceeds a predefined threshold θ_{index} when compared to the tag *tag*. $|R_e|$ and $|T_e^{tag}|$ are the number of elements in both R_e and T_e^{tag} respectively. Equation 1 finds all review tags which are similar to *tag* and computes the arithmetic mean of their similarity scores, weighted by the number of reviews. The motivation of multiplying the mean with the number of reviews for each entity is that the more reviews there are, the more statistically significant the degrees of truth become. That is why *SACCS* privileges the entities having more reviews.

Going back to the example in Figure 1, when the user submits a new utterance *"I want a restaurant with a romantic ambiance"*, *SACCS* extracts *romantic ambiance* from the utterance. Because this tag is unknown to *SACCS* , it adds it to the user tag history. Consequently, in the next indexing round, *SACCS* includes *romantic ambiance* to the index and computes its entity mappings along with their degrees of truth as has been explained above. This mechanism enables *SACCS* to adapt to new user needs.

3.2 Filtering

In this section, we provide details about how *SACCS* utilizes subjective tags to answer users subjective utterances.

Processing user utterances. Suppose the user submits a new utterance: *"I want an Italian restaurant in Melbourne that serves delicious food and has a nice staff"*. *SACCS* forwards this utterance to the underlying dialog system which finds the user intent (in this case *searchRestaurant*) and calls a corresponding search API (e.g. TripAdvisor, Yelp...). In this example, *SACCS* expects the API to return the set of restaurants that are in Melbourne and serve Italian food. We call this set $\mathcal{S}_{api}$. As mentioned before, neither the dialog system nor the search API understand subjective information in the utterance such as *delicious food* and *nice staff*, thereby ignoring them completely. *SACCS* extracts these tags from the utterance and use them to filter and rank $\mathcal{S}_{api}$ before showing the final results to the user.

Probing the index. If the subjective tags extracted from the user utterance exist in the index, the corresponding entities with their degrees of truth are directly taken from the index. For instance, in the previous utterance, *nice staff* exists in the index depicted in Table 1, and thus the matching set ("Vue du Monde", 0.92), ("Pizza Hut", 0.63) is extracted as is. We call this set $\mathcal{S}_{t1}$, where t1 = *"nice staff"*.

On the other hand, if the subjective tag is not found in the index, *SACCS* adds it to the user tag history as discussed in Section 3.1 and Figure 1 for later indexing. However, in order to provide a good answer to the user in real time, *SACCS* combines mappings of similar tags which are already in the index. To illustrate this, we go back to the previous example. *Delicious food* does not exist in the index of Table 1, but is similar to *good food* and *creative cooking*. In this case, *SACCS* calculates the union of the mappings corresponding to these two tags and multiply their degrees of truth by the similarity score of *delicious food* with each of the two subjective tags. Assume that:

$$s_1 = similarity(\textit{delicious food}, \textit{good food}) \quad (2)$$

$$s_2 = similarity(\textit{delicious food}, \textit{creative cooking}) \quad (3)$$

The set of entities that *SACCS* finds for *delicious food* is then $\mathcal{S}_{t2} = \{(\textit{"Vue du Monde"}, s_1 \times 0.89), (\textit{"Anchovy"}, s_1 \times 0.76 + s_2 \times 0.94), (\textit{"Pizza Hut"}, s_1 \times 0.82 + s_2 \times 0.34), (\textit{"Kazuki's"}, s_2 \times 0.85)\}$

After the construction of $\mathcal{S}_{api}$, $\mathcal{S}_{t1}$ and $\mathcal{S}_{t2}$, *SACCS* needs to aggregate the entities coming from the search API, plus the ones recovered from each subjective tag in the utterance. In other words, *SACCS* computes the **intersection** of these sets of entities according to Algorithm 1. It is worth noting that the function *search_api* takes the user utterance as input parameter and relies on the underlying dialog system and the search API to provide results filtered by objective attributes alone. On the other hand, the function *extract_tags* takes the user utterance as input parameter and returns the list of subjective tags using the extraction pipeline which we describe in Sections 4 and 5.

Algorithm 1 Filtering & Ranking

1: Let **u** be the user utterance
2: Let **index** be the inverted index
3: Let θ_{filter} be the similarity threshold
4: $\mathcal{S}_{api} \leftarrow search\_api(u)$
5: $tags \leftarrow extract\_tags(u)$
6: **for** t in tags **do**
7: **if** $t \in index.keys$ **then**
8: $\mathcal{S}_t \leftarrow index[t]$
9: **else**
10: $\mathcal{S}_t \leftarrow \bigcup_{tag \in index.keys} \{\text{index[tag]}\}$ such that $similarity(t, tag) > \theta_{filter}$
11: $\mathcal{R} \leftarrow \bigcap_{t \in tags} \{\mathcal{S}_{api}, \mathcal{S}_t\}$
12: **Return** $sort(aggregate\_scores(\mathcal{R}))$

3.3 Ranking

SACCS ranks the filtered set of entities according to their degrees of truth across all subjective tags. We identify two situations for ranking.

One subjective tag. If the user expresses a single subjective filter in her utterance, the ranking is straight forward. *SACCS* sorts the entities according to their degrees of truth in descending order, so that the top results are the ones whose reviews strongly mention the subjective tag.

Many subjective tags. In this case, *SACCS* has a separate set of entities with their degrees of truth for each subjective tag. However, an entity can belong to many of such sets. Thus, before ranking becomes feasible, *SACCS* must aggregate the degrees of truth for each entity across all subjective tags. Aggregation is done via computing the arithmetic mean over all tags. We also experimented with other aggregation methods such as the product or min operators, but the arithmetic mean works better in practice. *SACCS* then sorts the entities in descending order. Algorithm 1 combines the filtering and ranking stages. In line 12, the function *aggregate_scores* computes the arithmetic mean of degrees of truth across the tags.

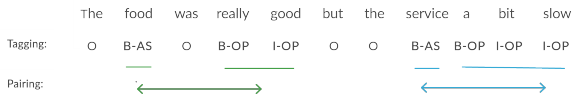


Figure 2: Token Tagging and Pairing

As mentioned before, a subjective tag is the concatenation of two terms: the **aspect** term and the **opinion** term. Following previous effort [31], we formulate the task of extracting subjective tags from a given input sentence as a two-stage process: tagging and pairing, as illustrated in Figure 2. Each word in the sentence is first tagged as being an aspect (AS), an opinion (OP) or neither (O). Then, every aspect term gets paired with its corresponding opinion term to build the set of subjective tags from the input sentence. In the following, we describe the techniques we propose for tagging and pairing tasks. We describe tagging in Section 4 and pairing in Section 5.

4 TAGGING

We denote by r_i a review sentence which consists of a sequence of tokens $r_i = \{w_{i1}, w_{i2}, ..., w_{in}\}$. We use the IOB encoding scheme [47] with the following classes: B-AS (Beginning of Aspect), I-AS (Inside of Aspect), B-OP (Beginning of Opinion), I-OP (Inside of Opinion) and O (Outside). The set of tags is thus L = {B-AS, I-AS, B-OP, I-OP, O}. The objective of tagging is to classify each token w_{ij} in the sentence r_i, into a class $c_{ij} \in L$. The components of *SACCS* 's tagging model are detailed below.

4.1 Baseline for the Tagging Pipeline

Figure 3 depicts the base architecture for tagging words into aspects and opinions. We use BERT [7], the recently-developed language model, as the embedding layer thanks to its proven superior quality when compared to other embedding models [7]. As illustrated in Figure 3, BERT embeddings serve as input to the Bidirectional LSTM (BiLSTM) layer [16], which encodes the past context (all words prior to any given word in the sentence) and the future context (all words following a given word) of each word. Following [8, 35], we encode the text sequence from both left to right (forward) and right to left (backward). We then concatenate the resulting representations to form the final output of the BiLSTM.

Finally, the BiLSTM output flows to the Conditional Random Field (CRF) layer [28], which is paramount to encode dependencies between the different labels of L. For example, I-OP cannot follow I-AS in the label sequence. More generally, I-AS (or I-OP) must either follow B-AS or I-AS (B-OP or I-OP). Given an input sequence $z = \{z_1, z_2, ..., z_n\}$, CRFs effectively utilize correlations between labels to predict the best label sequence $y = \{y_1, y_2, ..., y_n\}$. Formally, the conditional probability function of CRFs is given by:

$$P(y|z, W, b) = \frac{\prod_{i=1}^{n} \psi_i(y_{i-1}, y_i, z)}{\sum_{y' \in Y(z)} \prod_{i=1}^{n} \psi_i(y'_{i-1}, y'_i, z)} \quad (4)$$

where $Y(z)$ denotes the set of possible labels for the sequence z and $\psi_i(y_{i-1}, y_i, z) = exp(W^T_{y',y} z_i + b_{y',y})$ are potential functions to be learned with $W_{y',y}$ and $b_{y',y}$ being the weight and bias vectors respectively. Decoding (i.e. solving the tagging task using

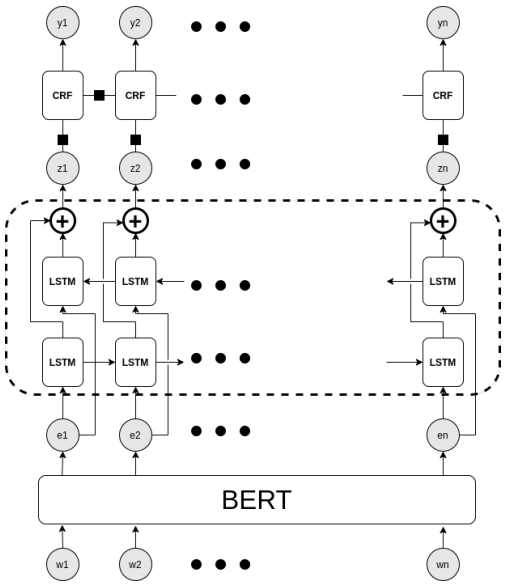


Figure 3: Sequence tagging model based on BERT + BiLSTM + CRF

a CRF layer) consists in finding the best sequence of labels y that maximizes the log-likelihood given the input sequence z:

$$y^* = \underset{y' \in Y(z)}{\text{argmax}}\, P(y'|z, W, b) \tag{5}$$

In this work, we use linear-chain CRFs, where only interactions between two successive labels are taken into consideration. We also adopt the Viterbi algorithm [10] along with beam search for efficient decoding of the label sequence.

4.2 Extending the Baseline with Domain Adaptation

In *"La carte of this restaurant is a killer"*, *SACCS* should be able to tag *la carte* as an aspect and *a killer* as an opinion. However, opinions are mostly adjectives whereas *a killer* is a noun, thereby *SACCS* might fail to recognize it as an opinion, or even mark it as an aspect. Moreover, *la carte* is a rare word in the english vocabulary, thus the tagger might not understand the word altogether. This limitation is largely due to the fact that BERT has been pre-trained on general Wikipedia articles [7]. As a consequence, it does not know that *a killer* is a widely used idiom in the restaurant jargon to characterize something as overly good. It also ignores that *la carte* in this case means *the menu*, which is an important aspect to be extracted. Hence, standard BERT embeddings are blind to the domain and may hinder the tagging performance of *SACCS* .

To make the embeddings more domain-aware, we follow the guidelines of [58] who post-trained BERT on domain-specific review corpora in order to make it understand opinion text rather than generic Wikipedia articles. We use reviews about restaurants as a post-training dataset in our case. [58] also added another fine-tuning iteration to make BERT aware of the task (e.g. aspect/opinion extraction), but on out-of-domain data. We find that using domain knowledge alone works better in our case than when leveraging both domain- and task-awareness. Experiments show that domain adaptation adds up to 2.93 F1 score points over the baseline.

4.3 Adversarial Learning for Dealing with Language Expressiveness

Natural language is very nuanced, and introducing subtle changes to the input sentences can change the meaning dramatically. For example, adding *not* before the verb or changing *always* with *never* reverse the meaning of the sentences completely. Unfortunately, such changes happen frequently when using the trained model with new sentences, unseen during training. This is particularly alarming when the changes are subtle, or insignificant when assessed by a human evaluator, for example changing a word with its synonym. However, word embeddings do not always align with human perception. For instance, two synonymous words might be far apart in the embedding space [9, 21, 40]. Even if they are close to each other, the tiny distance between the embeddings can be enough to mislead the trained model [38]. Adversarial examples have long been used to make trained models robust against small input differences and perturbations (noise). It has been shown to provide additional regularization capabilities beyond that brought by the use of dropout alone [13].

We leverage adversarial learning to enhance the robustness of *SACCS* 's tagger against input noise. We generate adversarial examples that are close to the original inputs and that should share the same label sequence (i.e. aspect/opinion tags), yet are specifically designed to fool the model into tagging them otherwise. The creation of these adversarial inputs is enabled by the introduction of small *worst case* perturbations bounded by a chosen perturbation set, to decrease the model's ability to predict correctly. The tagger is then trained on a mixture of clean and adversarial examples to enhance its stability and robustness against potential input perturbation. The objective function is thus the following:

$$\underset{\theta}{\text{Min}}[\alpha.l(h_\theta(x), y) + (1-\alpha).\underset{\delta \in \Delta(x)}{\text{Max}}\, l(h_\theta(x+\delta), y)] \tag{6}$$

where h_θ is the tagging model with θ being the corresponding parameters. l is the loss function and $\Delta(x)$ is the set of perturbations allowed for the input sequence x. In this work, we use the l_∞ ball: $\Delta(x) = \{\delta : ||\delta||_\infty < \epsilon\}$ where ϵ is a hyperparameter to be tuned. Equation 6 assumes the perturbations to be applied directly on the embeddings as has been done in [38]. Solving such an objective function exactly is intractable in complex networks. Consequently, by leveraging Danskin's theorem [6], we can first solve the inner maximization independently to find δ^* that maximizes the adversarial loss, and then adding δ^* to the input to solve the outer minimization objective.

$$\delta^* = \underset{||\delta||_\infty < \epsilon}{\text{argmax}}\, l(h_\theta(x+\delta), y) \tag{7}$$

$$\underset{\theta}{\text{Min}}[\alpha.l(h_\theta(x), y) + (1-\alpha).l(h_\theta(x+\delta^*), y)] \tag{8}$$

Finding an exact solution for δ^* is also an intractable problem for complex models. We approximate δ^* by assuming a *linear* tendency for the adversarial loss inside the norm-ball. We thus use the Fast Gradient Sign Method (FGSM) suggested by [13] to find a decent solution in an efficient way. The computation of δ^* is given by:

$$\delta^* = \epsilon.sign(g) \tag{9}$$

where $g = \nabla_\delta l(h_\theta(x+\delta), y)$. In Equation 8, the first loss is the clean loss, while the second loss represents its adversarial counterpart. The parameter α denotes how much weight we

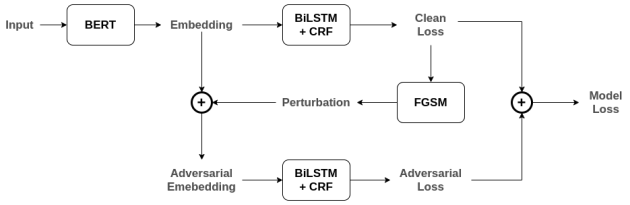


Figure 4: Architecture for Adversarial learning using BERT

give to the adversarial example with respect to the original one. Figure 4 illustrates the entire adversarial learning component.

5 PAIRING

In Figure 2, *food* is paired with *really good*, and *service* with *a bit slow* [3] in order to create the corresponding subjective tags. Most previous work [55, 56] employ simple heuristics such as word distance to pair aspects and opinions. However, such techniques fall short of the expected accuracy especially on complex and tricky input sentences. For example, the opinion *professional* would be wrongfully paired with the aspect *decor* in the review *"The staff is friendly, helpful and professional. The decor is beautiful"* when relying on word distance alone, because *professional* is closer to *decor* than to *staff*.

In this section, we first describe the two novel heuristics that we propose to pair aspects with opinions (Section 5.1). Although these heuristics can be directly used as an unsupervised pairing model, in Section 5.2, we discuss how they are used in a supervised model.

5.1 Pairing Heuristics

We design two types of unsupervised heuristics for pairing. The first category is based on constituency parse trees [24, 25, 41] while the second utilizes the attention mechanism [17].

Heuristic based on parse trees. The first method is a rule-based method. The intuition behind it is that associated aspects and opinions should be close to each other in the parse tree of the input sentence. We start by building the parse tree and then apply a greedy strategy that maps every aspect term to the "closest" opinion term in the parse tree. Given that a single aspect can be mapped to multiple opinions [4], we use this heuristic twice: from aspects to opinions and then from opinions to aspects. For example, in *"The staff is friendly and professional"*, *friendly* is closer to *staff* than *professional* is in the parse tree. Hence, the first version outputs the pair (*staff*, *friendly*). On the other hand, the second run starts from opinions and looks for the closest aspect. It would thus give the pairs (*staff*, *friendly*) and (*staff*, *professional*).

Heuristic based on BERT attention heads. The idea behind using BERT attention heads for pairing is motivated by the need to assign relevance scores to aspects and opinions. Ideally, we want each aspect term to focus more on its corresponding opinion (high relevance score) and ignores the rest (low relevance scores). Attention can be leveraged to approximate relevance. First introduced to enhance neural machine translation

[3] A multi-word aspect (or opinion) is regarded as a single aspect (opinion) term
[4] The reverse also applies: An opinion term can be paired with multiple aspects as well

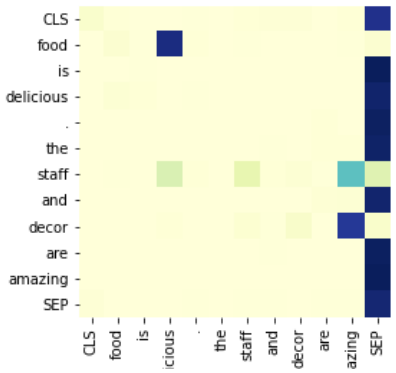


Figure 5: BERT attention head for pairing aspect and opinion terms

[3] and later adapted to nearly every other NLP task, attention is a mechanism to assign importance values to every token in the sequence given a query term. We say that the query term attends to the tokens which have the highest attention scores. In our case, the query term is the aspect term, and the sequence is the input sentence. Using this method, the goal is to distribute the attention of every aspect term so that it attends to the rightful opinion (that of the highest attention).

Fortunately, BERT is an attention-based model, and we have it already trained on aspect/opinion extraction as explained in Section 4. We hypothesize that, while learning the downstream task of tagging aspects and opinions, BERT leverages its attention heads in a way that makes aspects attend to the rightful opinions, and vice versa. Figure 5 confirms our hypothesis. It illustrates one attention head of BERT. Each row in the figure is the attention distribution of the corresponding word over the entire input sequence; the darker the color, the higher the attention. In the figure, *food* is darkest at *delicious*, meaning that *food*'s attention to *delicious* is very high. In the same spirit, both *staff* and *decor* attend to *amazing*. Thus, BERT attention heads act as simple no-training-required classifiers that, given an aspect, output the most attended to opinion. We find that BERT heads capture various linguistic properties, some of which correspond remarkably well to the notion of pairing aspects with opinions. The best head we found for pairing has an accuracy of 82.62% on the pairing test set (Section6.4), which is excellent given the quasi-none effort this method needs.

5.2 Supervised Learning-based Approach for Pairing

We also provide a supervised learning alternative to the problem of pairing. We formulate our pairing objective as a classification problem. Given an input sentence s_i (e.g., *"The food is delicious and the staff are friendly"*) and a short phrase p_i (e.g., *"delicious food"*)[5], the classifier classifies p_i as being a correct extraction from s_i or not. To use this classifier with *SACCS* , we first use the tagger to extract aspects and opinions from s_i. We then construct all possible pairs from the sets of aspects and opinions regardless of their soundness. For example, suppose we have *food* and *staff* as aspects, and *delicious* and *friendly* as opinions. The list of all possible pairings is: P_{all} = [*"delicious food"*, *"delicious staff"*,

[5] In the context of this work, these short phrases are subjective tags

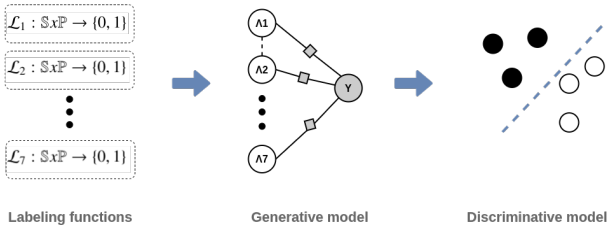


Figure 6: Data Programming pipeline for pairing

"friendly food", *"friendly staff"*]. We feed s_i with each pair from P_{all} into the classifier, and consider it as a correct extraction if the classifier returns a positive label.

We use data programming [2, 49] in order to create the dataset necessary to train such a classifier. The entire pipeline is illustrated in Figure 6. First, a set of labeling functions [2, 48, 49] use the heuristics described in Section 5.1 in order to independently assign a label to every (s_i, p_i) pair. These labeling functions are considered weak supervision sources.

The second step in the pipeline aggregates the labels from the labeling functions to construct a single overall label for every (s_i, p_i) pair, based on agreements and disagreements between labeling functions. This is generally achieved with generative models which, by aggregating enough datapoints, end up creating a decent labeled training dataset. Finally, we use this dataset to train a discriminative model. In our case, it's the classifier discussed above. It is important to note that, in our case, we have a working solution for pairing aspects with opinions at each step of the pipeline. However, experiments show that committing to the entirety of the pipeline and using the discriminative model (and thus the supervised model) drives a considerable boost in pairing accuracy when compared to the unsupervised methods.

In the following, we describe the labeling functions, the generative and discriminative models we use in the supervised learning-based pairing pipeline.

Labeling functions for the pairing pipeline. A labeling function in *SACCS* 's pairing module has the same interface as the classifier, i.e. expects a sentence s_i and a phrase p_i as input, and outputs a binary label telling whether p_i is a legit extraction from s_i. All labeling functions are based on the heuristics presented in Section 5.1. To transform each heuristic H_j into a labeling function L_j, we follow the procedure below:

(1) Extract all aspects and opinions from s_i using *SACCS* 's tagger.
(2) Use H_j to find the pairs $P_{H_j^i}$ as detailed in Section 5.1.
(3) If the short phrase p_i belongs to the set of constructed pairs $P_{H_j^i}$, output 1. Otherwise, return 0.

We use seven different labeling functions: two are based on the parse tree method (the first from aspects to opinions, the second the other way around) while the remaining five rely on BERT attention scores employing different heads. The choice of attention heads has been made after a qualitative analysis.

Generative model for the pairing pipeline. We use the generative model proposed by Snorkel [48] in our pipeline. Snorkel is a data programming framework that integrates the noisy signals of multiple labeling functions to estimate the true label class[48]. Snorkel offers two mechanisms for aggregation. The simplest is a majority vote model where each labeling function is regarded as an independent voter. The chosen label for each datapoint is the most agreed upon by labeling functions. The other method incorporates statistical properties of labeling functions such as accuracies and correlations. Snorkel then trains a probabilistic graphical model to *generate* the true labels without access to ground truth data. Training is based on agreements and disagreements between the different labeling functions as dictated by data programming. Although the authors of Snorkel state that the probabilistic generative model works better in practice than the majority vote, we found the latter to be more accurate.

We can directly use the generative model to extract subjective tags from review sentences. However, a better use of data programming lies in the automatic creation of labelled training data to train a subsequent discriminative model. The advantages of doing so are twofold: First, the discriminative model generalizes beyond the scope of examples fed to the labeling functions. Second, the discriminative model is faster to execute because the generative model loops through all labeling functions and aggregates their outputs, whereas the discriminative model only uses one forward pass in case of neural networks.

Discriminative model for the pairing pipeline. We train a simple two-layer neural network with a sigmoid activation function. We encode s_i and p_i using BERT embeddings. We train the classifier with the training data that has been automatically created with the procedure explained in the previous sections. Our experiments confirm that the discriminative model outperforms the generative one, as has been found in [48].

6 EXPERIMENTS

We first begin by showing our experimental settings before describing the experiments that we conducted. The first set of experiments evaluates the overall performance of *SACCS* and compares it to two baselines (Section 6.2). We then move to assess the quality of *SACCS* components. We evaluate the sequence tagger in Section 6.3 and the pairing mechanism in Section 6.4.

6.1 Experimental Settings

Datasets. We apply *SACCS* to the domain of restaurants whose online reviews we get from the publicly available Yelp Dataset [1]. Since it covers a wide array of businesses, we filter it to only keep reviews about Italian restaurants in Montreal, resulting in 280 entities (restaurants) with 7061 reviews. To train the aspect/opinion tagger, we use the training dataset created by [31] that contains 800 sentences, wherein each token is accompanied by its label. For pairing, we use the same training dataset as in [31] but without the labels since we augment the data and infer the labels with Snorkel [48].

Processing. We implemented *SACCS* in Python using standard packages such as PyTorch [42] for neural networks, HuggingFace transformers library [57] for BERT, NLTK [34] for textual preprocessing and Scikit-Learn [43] for evaluation metrics. In order to incorporate domain knowledge into BERT, we directly use the models for restaurants published on Huggingface community hub by [58]. For adversarial training, we fix the value of α to 0.5 (Equation 8) while we vary ϵ between {0.1, 0.2, 0.5, 1.0, 2.0}.

6.2 Comparing *SACCS* with Baselines

In this experiment, we evaluate the overall performance of *SACCS* , and then compare it to two strong baselines. The evaluation works as follows: we first prepare a set of subjective tags as a test set. Each system that we want to evaluate takes the tags as input and returns an ordered list of results, sorted by their degree of relevance with respect to the subjective tags. The result of each system is then compared to the ideal ordering of entities. The system whose ordering is "closest" to the ideal one is deemed the best. We apply this experiment to the domain of restaurants.

Preparing subjective tags. Since there is no benchmark for subjective tags, we had to create our own. [39] identified the most important features restaurant seekers consider when choosing a restaurant. These features include *"delicious food"*, *"creative cooking"*, *"varied menu"*, *"romantic ambiance"*... We chose 18 of them to serve as our subjective tags for testing purposes. We then construct combinations of these tags by uniform random sampling. Each combination will form a potential subjective user utterance. For example, if random sampling puts together the tags *"clean plates"* and *"quick service"*, it works as if a user gave the following utterance to the system: *"I am looking for a restaurant that delivers a quick service with clean plates."*. The number of tags per combination depends on the level of difficulty of the query (utterance). In this experiment, we set 3 levels of difficulty: Short with either 1 or 2 tags; Medium with 3 or 4; Long with 5 or 6 tags. Each set (level of difficulty) contains 100 queries (combinations).

Evaluation metrics. To measure how well the entities returned by *SACCS* and the baselines satisfy the queries in the test set, we use the well-known Normalized Discounted Cumulative Gain (NDCG) [5] which is a measure of ranking quality. Formally, this metric computes the quality of a ranked list and divides it by that of the ideal ordering, thus giving a score between 0 and 1, the higher the better. For illustration purposes, assume that subjective query Q has n subjective tags: $Q = \{q_1, q_2, ..., q_n\}$ and that we input Q to *SACCS* . The latter returns a list of top-k entities $E = \{e_1, e_2, ..., e_k\}$. We define $sat(q_i, e_j) \in [0, 1]$ as the degree with which entity e_j satisfies the subjective tag q_i. The NDCG score is computed as follows:

$$DCG(Q, E) = \sum_{j=1}^{k} (2^{\frac{1}{m}\sum_{i=1}^{m} sat(q_i, e_j)} - 1)/log_2(j+1) \quad (10)$$

$$NDCG(Q, E) = DCG(Q, E)/iDCG(Q) \quad (11)$$

Intuitively, a highly relevant entity (that with $sat(q_i, e_j)$ scores close to 1) should be at the top in order for the DCG to be high. iDCG in Equation 11 corresponds to the DCG score of the ideal ordering. It is fairly easy to get the iDCG as it is only a matter of sorting the entities with respect to the sum of their $sat(q_i, e_j)$ scores and then computing the DCG. Finally, we take the arithmetic mean over all queries to compute the quality of the entire test set.

Ground truth. We obtain the ground truth $sat(q_i, e_j)$ of subjective tag q_i and entity e_j via crowdsourcing. We give each worker a tag q_i and one review r_j^k from the set of online reviews corresponding to entity e_j. The crowdsourcing task is to inspect the review r_j^k and tell whether it mentions the tag q_i or not. The worker must assign each pair of review/tag a relevance score among the following: 0 for no relevance, $\frac{1}{3}$ for weak relevance, $\frac{2}{3}$ for strong relevance and 1 for perfect relevance. As an example, given the review sentence *"The food is very delicious but the service is terrible"*, the tag *great food* should be marked as perfectly relevant, *nice decor* not relevant while *slow service* as weakly relevant because the slowness of the service is somewhat related to it being terrible. For each review/tag pair, we ask three different workers to provide labels, from which we take the majority vote, resulting in $sat(q_i, r_j^k)$ relevance scores. To obtain $sat(q_i, e_j)$, we take the mean of $sat(q_i, r_j^k)$ across the reviews of the same entity e_j. The crowdsourcing experiment has been conducted on Yandex Toloka platform[6].

Table 2: Comparing *SACCS* to baselines

System	Short	Medium	Long
IR	0.829	0.896	0.916
SIM - 1 att	0.828	0.886	0.907
SIM - 2 atts	0.837	0.891	0.909
SACCS - 6 tags	0.815	0.874	0.896
SACCS - 12 tags	0.825	0.882	0.902
SACCS - 18 tags	**0.854**	**0.911**	**0.928**

Baselines. We compare *SACCS* to two baselines: an Information Retrieval (IR) system and a custom simulation (SIM). The IR baseline uses Okapi BM25 [5] retrieval model. We follow the work of [11] and add the capability to expand the terms of the query into synonymous and related terms, as well as select the best query combination method they found to make the IR system more competitive.

SIM represents what a determined and tireless user can get from Yelp or other similar online services. Because these services provide a set of queryable attributes (such as NoiseLevel, Ambiance or GoodForGroups), the user might filter the search results with the attributes she thinks closely resemble her subjective preferences. For example, if she is interested in quiet restaurants, she can set the attribute NoiseLevel to *calm* and the attribute GoodForGroups to *False* in Yelp's interface. She can also rank the results by star rating. SIM is a simulation of such behavior. We assume that the user can choose one or two attributes from Yelp's interface at a time. SIM computes all possible combinations of attribute values and selects the one that maximizes the NDCG score, thus finding the best top-k results that satisfy the subjective queries. It's needless to say that SIM constitutes a very strong baseline to compare *SACCS* against.

Comparison and analysis. Table 2 reports the NDCG scores of the three systems on the test set. Each column corresponds to the level of difficulty, that is Short, Medium or Long. The first row shows the quality of the IR system. The following two lines are variations of SIM using only one attribute, or a combination of two separate attributes. The last 3 rows describe the performance of *SACCS* , each time with a different number of subjective tags present in the index. This is to simulate the adaptive capability of *SACCS* as interactions with users unfold.

In all difficulty levels, *SACCS* outperforms the information retrieval system with a margin between 1.2% and 2.5%. This is not surprising because the IR system is based on keywords and looks for exact match whereas *SACCS* models subjective

[6] https://toloka.yandex.com

Table 3: Dataset Descriptions with number of sentences for train and test

Dataset	Description	Train	Test	Total
S1	SemEval-14 Restaurants	3041	800	3841
S2	SemEval-14 Electronics	3045	800	3845
S3	SemEval-15 Restaurants	1315	685	2000
S4	Booking.com Hotels	800	112	912

attributes with subjective tags. Table 2 shows that *SACCS* is superior to keyword-based systems even when the latter are bulked with query expansion and adequate predicate aggregation techniques. On the other front, SIM simulates the behavior of a determined user that runs through all possible combinations of queryable attributes that online services such as Yelp offer. To make the evaluation challenging, we take the combination that maximizes the NDCG score, thus reflecting the best result a user can have when interacting with Yelp's interface. As shown on the table, considering two attributes yields better results than one attribute, but with diminishing returns. That is why we don't bother searching the space of more than two attributes, which adds a non-negligible amount of computation. *SACCS* outruns SIM with 2 attributes by a margin between 1.7% and 2.0%.

Even with a small number of tags in the index, the performance of *SACCS* is comparable to that of IR or SIM. This is especially the case at the initialization of the index, where it finds itself nearly empty (in Table 2, the index contains 6 tags only). However, as *SACCS* interacts with users, it extracts new subjective tags from user utterances and adds them to the index in a dynamic and adaptive way. This experiment demonstrates that adding more tags to the index improves the overall accuracy (improvement between 3.2% and 3.9%), and confirms that *SACCS* adapts to new user needs.

We also observe that, for all three systems, accuracy increases with a higher number of subjective criteria. We hypothesize that with more subjective tags, the list of restaurants which verify all the subjective filters shrinks, leading to a lower margin for error in all systems; thus a higher NDCG score. Nonetheless, *SACCS* is still the best no matter the number of subjective tags to be considered. We also observe that the largest improvement happens with short queries (1 or 2 subjective tags therein). This result reinforces the integration of *SACCS* to task-oriented dialog systems where utterances are short and usually span a small number of subjective filters.

6.3 Sequence Tagging Evaluation

We show that the aspect and opinion tagger of *SACCS* is of better quality than that of state of the art, especially when the training dataset is small. We evaluate the sequence tagging model with 4 different datasets summarised in Table 3. The first three datasets are from SemEval competitions: SemEval 2014 Task 4 (Restaurants and Electronics)[45] and SemEval 2015 Task 12 (Restaurants)[44]. Each dataset contains a set of sentences where each token is labeled as being an aspect, an opinion or neither, following IOB coding scheme [47]. The original SemEval datasets contain labels for aspects only. However, we use the versions of [31, 55, 56] who added labels for opinions to the original sentences. The last dataset has been created and labeled by [31]. The goal of this experiment is to compare *SACCS* 's tagger with the strongest previous works in the literature, using datasets of different sizes and domains as well.

Table 4: Evaluation of aspect/opinion tagger

Models	S1	S2	S3	S4
OpineDB	81.82	75.44	72.30	67.41
OpineDB + DK	83.06	75.42	73.86	69.64
Adversarial ($\epsilon = 0.1$)	81.23	76.56	**74.63**	70.16
Adversarial ($\epsilon = 0.2$)	83.46	**76.97**	73.64	**72.34**
Adversarial ($\epsilon = 0.5$)	**84.43**	75.36	72.28	70.32
Adversarial ($\epsilon = 1.0$)	82.80	67.50	73.47	70.38
Adversarial ($\epsilon = 2.0$)	82.93	71.39	73.27	68.42

Other extraction tasks such as Named Entity Recognition (NER) [51] employ F1 to measure the quality of tagging. In the same spirit, Table 4 reports F1 scores of the extraction quality. For an aspect (or opinion) to be counted as correctly extracted, it needs to match the exact terms present in the ground truth. We compare our tagger against two strong baselines: OpineDB's tagger [31] which is a BERT-based solution that outperformed previous works in the literature [55, 56] and currently enjoys state of the art performance. We also enhance OpineDB's tagger with the domain-specific fine-tuning strategy suggested by [58] to make it even stronger (**OpineDB + DK** in Table 4). We evaluate our adversarial tagging model with different sizes of perturbations (ϵ values) as shown in Table4, but we fix $\alpha = 0.5$ (Equation 8) across all runs. The models are trained for 15 epochs.

The adversarial tagging model beats state of the art performance in all four datasets with an improvement ranging from 1.53% to 4.93%. As shown in the table, fine-tuning BERT with domain knowledge (DK) improves the performance by up to 2.23%, confirming the findings of [58] on aspect-based sentiment analysis. However, the boost of domain fine-tuning is not enough to outperform the adversarial training component, motivating the integration of adversarial examples in deep learning models. We also note that the adversarial component works better for smaller datasets (S4). We believe this is due to the regularization capabilities adversarial training provides as a counter-measure against overfitting beyond what is already ensured with dropout. We also notice that the tagging model performs best with lower perturbation sizes ($\epsilon \in \{0.1, 0.2, 0.5\}$), in which case the adversarial examples remain "closer" to the original ones, in contrast to large perturbation sizes ($\epsilon \in \{1.0, 2.0\}$) that can lead the model to exhibit poor accuracy. The issue of large ϵ values is especially noticeable with the Electronics dataset (S2) where $\epsilon = 1.0$ makes the adversarial model worse than OpineDB's baseline. We hypothesize that, since the Electronics dataset contains many technical terms such as brand names and numerical references, adding slight perturbations can change the meaning of terms completely while keeping the same labels, leading to the model's poor performance.

These results are very promising in the context of task-oriented dialog systems. Since chatbots should cover a wide array of domains, they need to be trained and fine-tuned for every single one of them. This task implies the creation of various datasets that are large enough to ensure a decent learning. Fortunately, Table 4 shows that our tagging model is efficient even with small training data (S4), thus eliminating the need to build large and costly datasets.

Table 5: Evaluation of the pairing models

Models	Accuracy	Precision	Recall	F1
OpineDB	83.87	/	/	/
lf_bert_7:10	82.62	95.02	78.36	85.89
lf_bert_3:10	74.56	91.54	68.66	78.46
lf_bert_3:8	68.26	91.76	58.21	71.23
lf_bert_4:6	75.82	93.00	69.40	79.49
lf_bert_8:9	77.33	94.95	70.15	80.69
lf_tree_op	74.06	92.31	67.16	77.75
lf_tree_as	76.07	91.00	71.64	80.17
Majority Vote	84.10	97.20	78.70	87.00
Probablistic Model	82.40	**98.10**	75.40	85.20
Discriminative	**86.90**	92.52	**87.69**	**90.04**

6.4 Pairing Evaluation

In this section, we evaluate the accuracy of the pairing model. We use the test benchmark created by [31] and employed to conduct their own experiments. Each test example consists of a review sentence (e.g., *"The food is delicious and the staff is helpful"*), a tag (*"delicious staff"*) and the label is whether the tag is a correct extraction from the review sentence. The test set contains 397 sentences with a fairly equal amount of positive and negative examples. We compare the accuracy of *SACCS* 's pairing model with that of [31] in Table 5. To highlight the effectiveness of data programming in the context of pairing and motivate the use of both generative and discriminative models, we also assess the quality of every step in the data programming pipeline presented in Section 5.2. Thus, Table 5 reports the accuracy, precision, recall and F1 scores of all seven labeling functions that we used in our solution, both types of generative models (Majority vote and the probabilistic graphical model) and the supervised discriminative classifier.

We take the accuracy score of OpineDB pairing method directly from their paper [31] as we use the same test set. However, they do not report their precision, recall and F1 scores. In Table 5, lf_bert_**l**:**h** corresponds to the labeling function that is based on BERT using the attention head number **h** at layer **l**. lf_tree_op is the labeling function that uses the parse tree and that goes from each opinion to its closest aspect. lf_tree_as travels from aspects to opinions. We train the model with Booking.com dataset for hotels.

SACCS 's pairing model outperforms that of [31] by a margin of 3.03% in accuracy. This result confirms the effectiveness of data programming and weak supervision, and shows that really robust and efficient deep learning models can be designed with little effort and much less resources rather than relying on costly manual annotation. In Table 5, the labeling functions have different accuracies but they all suffer from low recall. We believe this phenomenon is due to the fact that labeling functions are simple heuristics in the first place, and thus fail to cover the entirety of the input space. On the other hand, they all enjoy very high precision, ranging from 91.00% to 95.02%. This insight sheds some light on the nature of our labeling functions and suggests to direct future work on designing heuristics that are less-precise but wide-reaching in order to balance precision and recall ratios. The generative models in Table 5 inherit the high precision of the labeling functions, with the probabilistic model scoring an outstanding 98.10%. However, they also drag the low recall, but are better in general than labeling functions when taken separately. This is due to the nature of generative models which maximize the benefits of labeling functions while minimizing their risk by combining and integrating their respective labels. Our findings support the original statements of [48]. Nevertheless, the experiment shows that the majority vote model surpasses the probabilistic graphical model in terms of accuracy, unlike what [48] reported. One explanation for this is that our labeling functions are already accurate enough and have comparable F1 scores, leading to similar votes. Thus, consensus should be relatively easier to reach, translating to better accuracy. Finally, we find that the discriminative model is the top scoring model in both accuracy, recall and F1, because it has been trained in a supervised fashion. Rather than depending on labeling functions to provide noisy labels, the discriminative model analyzes the feature space and generalizes its classification decisions to new and potentially unseen input; hence the high recall.

7 CONCLUSION

We proposed *SACCS* : a Natural Language Understanding module for task-oriented dialog systems which allows to recognize the subjective signals in user utterances and filter search results accordingly. *SACCS* is based on the inverted index data structure and mines subjective information from online reviews. We propose a novel subjective tag extraction pipeline that is robust against variations of natural language. We also propose two novel heuristics for pairing an aspect to an opinion. These heuristics aim to overcome the limitation of word-based distance approaches for pairing an aspect term to an opinion term. We also performed extensive evaluation of the proposed subjective tag extraction and pairing techniques. The performed experiments show that these techniques outperform existing approaches.

The advancements brought by *SACCS* are promising, albeit far from perfect. As future work, we plan to investigate the incorporation of search automata as a substitute for inverted indexes. Subjective digital assistants should be able to take into account user profiles and adjust their search and interaction behavior accordingly. We also plan to extend the robustness of the proposed techniques to cater for biased or fraudulent online reviews. For instance, a reviewer might have been paid by a business owner to write positive reviews about it, or negative reviews about its competitors. We have to differentiate between truthful and fake reviews in order to provide a transparent search experience for users. Finally, given the importance of thresholds in similarity assessments, it would be useful for *SACCS* to adjust these dynamically depending on the semantics of the subjective tags being compared.

ACKNOWLEDGEMENT

This work was partially supported by the PICASSO (IDEX/FEL/2018/01 - 18IA102UDL) project at LIRIS centre.

REFERENCES

[1] 2020. *MS Windows NT The Yelp Dataset.* https://www.yelp.com/dataset/
[2] Stephen H Bach, Bryan He, Alexander Ratner, and Christopher Ré. 2017. Learning the structure of generative models without labeled data. *Proceedings of machine learning research* 70 (2017), 273.
[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[4] Hongshen Chen, Xiaorui Liu, Dawei Yin, and Jiliang Tang. 2017. A survey on dialogue systems: Recent advances and new frontiers. *Acm Sigkdd Explorations Newsletter* 19, 2 (2017), 25–35.
[5] D Manning Christopher, Raghavan Prabhakar, and Schacetzel Hinrich. 2008. Introduction to information retrieval. *An Introduction To Information Retrieval* 151, 177 (2008), 5.

[6] John M Danskin. 2012. *The theory of max-min and its application to weapons allocation problems*. Vol. 5. Springer Science & Business Media.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. 2015. Transition-based dependency parsing with stack long short-term memory. *arXiv preprint arXiv:1505.08075* (2015).

[9] Manaal Faruqui, Jesse Dodge, Sujay K Jauhar, Chris Dyer, Eduard Hovy, and Noah A Smith. 2014. Retrofitting word vectors to semantic lexicons. *arXiv preprint arXiv:1411.4166* (2014).

[10] G David Forney. 1973. The viterbi algorithm. *Proc. IEEE* 61, 3 (1973), 268–278.

[11] Kavita Ganesan and Chengxiang Zhai. 2012. Opinion-based entity ranking. *Information retrieval* 15, 2 (2012), 116–150.

[12] Jianfeng Gao, Michel Galley, Lihong Li, et al. 2019. Neural approaches to conversational ai. *Foundations and Trends® in Information Retrieval* 13, 2-3 (2019), 127–298.

[13] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[14] Alon Y Halevy. 2019. The Ubiquity of Subjectivity. *IEEE Data Eng. Bull.* 42, 1 (2019), 6–9.

[15] Homa B Hashemi, Amir Asiaee, and Reiner Kraft. 2016. Query intent detection using convolutional neural networks. In *International Conference on Web Search and Data Mining, Workshop on Query Understanding*.

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[17] Dichao Hu. 2019. An introductory survey on attention mechanisms in NLP problems. In *Proceedings of SAI Intelligent Systems Conference*. Springer, 432–448.

[18] Minqing Hu and Bing Liu. 2004. Mining and summarizing customer reviews. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. 168–177.

[19] Minqing Hu and Bing Liu. 2004. Mining opinion features in customer reviews. In *AAAI*, Vol. 4. 755–760.

[20] K Indhuja and Raj PC Reghu. 2014. Fuzzy logic based sentiment analysis of product review documents. In *2014 First International Conference on Computational Systems and Communications (ICCSC)*. IEEE, 18–22.

[21] Sujay Kumar Jauhar, Chris Dyer, and Eduard Hovy. 2015. Ontologically grounded multi-sense representation learning for semantic vector space models. In *proceedings of the 2015 conference of the north American chapter of the Association for Computational Linguistics: human language technologies*. 683–693.

[22] Wei Jin, Hung Hay Ho, and Rohini K Srihari. 2009. A novel lexicalized HMM-based learning framework for web opinion mining. In *Proceedings of the 26th annual international conference on machine learning*, Vol. 10. Citeseer.

[23] Joo-Kyung Kim, Gokhan Tur, Asli Celikyilmaz, Bin Cao, and Ye-Yi Wang. 2016. Intent detection using semantically enriched word embeddings. In *2016 IEEE Spoken Language Technology Workshop (SLT)*. IEEE, 414–419.

[24] Dan Klein and Christopher D Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st annual meeting of the association for computational linguistics*. 423–430.

[25] Dan Klein and Christopher D Manning. 2004. Corpus-based induction of syntactic structure: Models of dependency and constituency. In *Proceedings of the 42nd annual meeting of the association for computational linguistics (ACL-04)*. 478–485.

[26] George Klir and Bo Yuan. 1995. *Fuzzy sets and fuzzy logic*. Vol. 4. Prentice hall New Jersey.

[27] Ari Kobren, Pablo Barrio, Oksana Yakhnenko, Johann Hibschman, and Ian Langmore. 2019. Constructing High Precision Knowledge Bases with Subjective and Factual Attributes. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2050–2058.

[28] John Lafferty, Andrew McCallum, and Fernando CN Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. (2001).

[29] Parisa Lak and Ozgur Turetken. 2014. Star ratings versus sentiment analysis–a comparison of explicit and implicit measures of opinions. In *2014 47th Hawaii International Conference on System Sciences*. IEEE, 796–805.

[30] Fangtao Li, Chao Han, Minlie Huang, Xiaoyan Zhu, Ying-Ju Xia, Shu Zhang, and Hao Yu. 2010. Structure-aware review mining and summarization. In *Proceedings of the 23rd international conference on computational linguistics*. Association for Computational Linguistics, 653–661.

[31] Yuliang Li, Aaron Feng, Jinfeng Li, Saran Mumick, Alon Halevy, Vivian Li, and Wang-Chiew Tan. 2019. Subjective databases. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1330–1343.

[32] Bing Liu. 2012. Sentiment analysis and opinion mining. *Synthesis lectures on human language technologies* 5, 1 (2012), 1–167.

[33] Pengfei Liu, Shafiq Joty, and Helen Meng. 2015. Fine-grained opinion mining with recurrent neural networks and word embeddings. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. 1433–1443.

[34] Edward Loper and Steven Bird. 2002. NLTK: the natural language toolkit. *arXiv preprint cs/0205028* (2002).

[35] Xuezhe Ma and Eduard Hovy. 2016. End-to-end sequence labeling via bi-directional lstm-cnns-crf. *arXiv preprint arXiv:1603.01354* (2016).

[36] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval*. Cambridge university press.

[37] Russell Mannion, Huw Davies, and Martin Marshall. 2005. Impact of star performance ratings in English acute hospital trusts. *Journal of Health Services Research & Policy* 10, 1 (2005), 18–24.

[38] Takeru Miyato, Andrew M Dai, and Ian Goodfellow. 2016. Adversarial training methods for semi-supervised text classification. *arXiv preprint arXiv:1605.07725* (2016).

[39] Luiz Rodrigo Cunha Moura, Gustavo Quiroga Souki, et al. 2017. Choosing a Restaurant: Important attributes and related features of a consumer's decision making process. *Revista Turismo em Análise* 28, 2 (2017), 224–244.

[40] Nikola Mrkšić, Ivan Vulić, Diarmuid Ó Séaghdha, Ira Leviant, Roi Reichart, Milica Gašić, Anna Korhonen, and Steve Young. 2017. Semantic specialization of distributional word vector spaces using monolingual and cross-lingual constraints. *Transactions of the association for Computational Linguistics* 5 (2017), 309–324.

[41] Joakim Nivre and Mario Scholz. 2004. Deterministic dependency parsing of English text. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*. 64–70.

[42] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[43] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[44] Maria Pontiki, Dimitrios Galanis, Harris Papageorgiou, Suresh Manandhar, and Ion Androutsopoulos. 2015. Semeval-2015 task 12: Aspect based sentiment analysis. In *Proceedings of the 9th international workshop on semantic evaluation (SemEval 2015)*. 486–495.

[45] Maria Pontiki, Dimitris Galanis, John Pavlopoulos, Harris Papageorgiou, Ion Androutsopoulos, and Suresh Manandhar. 2014. SemEval-2014 Task 4: Aspect Based Sentiment Analysis. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014)*. Association for Computational Linguistics, Dublin, Ireland, 27–35. https://doi.org/10.3115/v1/S14-2004

[46] Chen Qu, Liu Yang, W Bruce Croft, Falk Scholer, and Yongfeng Zhang. 2019. Answer interaction in non-factoid question answering systems. In *Proceedings of the 2019 Conference on Human Information Interaction and Retrieval*. 249–253.

[47] Lance A Ramshaw and Mitchell P Marcus. 1999. Text chunking using transformation-based learning. In *Natural language processing using very large corpora*. Springer, 157–176.

[48] Alexander Ratner, Stephen H. Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *Proc. VLDB Endow.* 11, 3 (Nov. 2017), 269–282. https://doi.org/10.14778/3157794.3157797

[49] Alexander J Ratner, Christopher M De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. 2016. Data programming: Creating large training sets, quickly. In *Advances in neural information processing systems*. 3567–3575.

[50] Régis Saint-Paul, Guillaume Raschia, and Noureddine Mouaddib. 2005. General purpose database summarization. In *Proceedings of the 31st international conference on Very large data bases*. 733–744.

[51] Erik F Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050* (2003).

[52] Dilip Kumar Sharma, Rajendra Pamula, and DS Chauhan. 2020. A contemporary combined approach for query expansion. *Multimedia Tools and Applications* (2020), 1–27.

[53] Junfeng Tian, Zhiheng Zhou, Man Lan, and Yuanbin Wu. 2017. Ecnu at semeval-2017 task 1: Leverage kernel-based traditional nlp features and neural networks to build a universal model for multilingual and cross-lingual semantic textual similarity. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*. 191–197.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[55] Wenya Wang, Sinno Jialin Pan, Daniel Dahlmeier, and Xiaokui Xiao. 2016. Recursive neural conditional random fields for aspect-based sentiment analysis. *arXiv preprint arXiv:1603.06679* (2016).

[56] Wenya Wang, Sinno Jialin Pan, Daniel Dahlmeier, and Xiaokui Xiao. 2017. Coupled multi-layer attentions for co-extraction of aspect and opinion terms. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[57] Thomas Wolf, L Debut, V Sanh, J Chaumond, C Delangue, A Moi, P Cistac, T Rault, R Louf, M Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv, abs/1910.03771* (2019).

[58] Hu Xu, Bing Liu, Lei Shu, and Philip S Yu. 2019. Bert post-training for review reading comprehension and aspect-based sentiment analysis. *arXiv preprint arXiv:1904.02232* (2019).

[59] Qiang Ye, Rob Law, and Bin Gu. 2009. The impact of online user reviews on hotel room sales. *International Journal of Hospitality Management* 28, 1 (2009), 180–182.

[60] Lotfi A Zadeh. 1975. The concept of a linguistic variable and its application to approximate reasoning—I. *Information sciences* 8, 3 (1975), 199–249.

GeoBlocks: A Query-Cache Accelerated Data Structure for Spatial Aggregation over Polygons

Christian Winter Andreas Kipf* Christoph Anneser
Eleni Tzirita Zacharatou◇ Thomas Neumann Alfons Kemper
Technische Universität München MIT CSAIL* Technische Universität Berlin◇
{winterch, anneser, neumann, kemper}@in.tum.de kipf@mit.edu eleni.tziritazacharatou@tu-berlin.de

ABSTRACT

As individual traffic and public transport in cities are changing, city authorities need to analyze urban geospatial data to improve transportation and infrastructure. To that end, they highly rely on spatial aggregation queries that extract summarized information from point data (e.g., Uber rides) contained in a given polygonal region (e.g., a city neighborhood). To support such queries, current analysis tools either allow only predefined aggregates on predefined regions and are thus unsuitable for exploratory analyses, or access the raw data to compute aggregate results on-the-fly, which severely limits the interactivity. At the same time, existing pre-aggregation techniques are inadequate since they maintain aggregates over rectangular regions. As a result, when applied over arbitrary polygonal regions, they induce an approximation error that cannot be bounded.

In this paper, we introduce GeoBlocks, a novel pre-aggregating data structure that supports spatial aggregation over arbitrary polygons. GeoBlocks closely approximate polygons using a set of fine-grained grid cells and, in contrast to prior work, allow to bound the approximation error by adjusting the cell size. Furthermore, GeoBlocks employ a trie-like cache that caches aggregate results of frequently queried regions, thereby dynamically adapting to the skew inherently present in query workloads and improving performance over time. In summary, GeoBlocks outperform on-the-fly aggregation by up to three orders of magnitude, achieving the sub-second query latencies required for interactive exploratory analytics.

1 INTRODUCTION

Nowadays, the amount of geospatial data collected in cities is increasing rapidly, thanks to the widespread use of mobility applications such as Uber [53]. To analyze this data and make data-driven decisions, city officials and planners often rely on visualization frameworks that allow users to visualize data of interest at different spatial and temporal resolutions [4, 8, 41, 50, 53]. To generate common visualizations, such as heatmaps, visual tools perform *spatial aggregation queries* that partition the data over different polygonal-shaped regions and then compute summarized aggregate information for each region. To support *exploratory analyses*, visual tools must provide interactive response times as high latency reduces the rate at which users make observations, draw generalizations, and generate hypotheses [22]. However, the sheer size of the data combined with the complexity of spatial queries prohibit interactivity, which severely limits analyses. As shown in [28], current tools operating over raw geospatial data cannot produce results fast enough for interactive analysis.

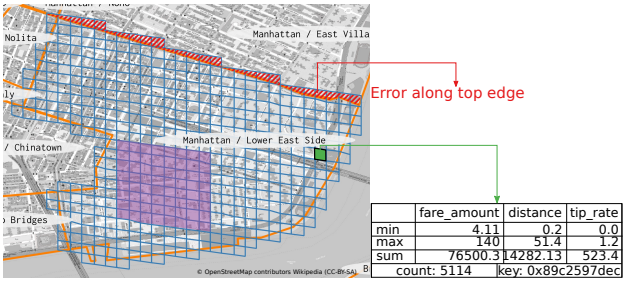


Figure 1: Cell covering (blue) of the Lower East Side (border in orange) with bounded error (red), a cell aggregate (green), and a cached commonly queried region (purple).

On the bright side, interactive analyses are often repetitive in nature. Analysts, for example, typically run multiple aggregate queries for the same area (e.g., the city center) in a sequence, changing only the aggregate function (e.g., count, sum) or the data attribute over which the aggregation is performed. Furthermore, they often focus on certain geospatial regions during their analysis. They might, for example, iteratively resize the boundary of the spatial region of interest, extracting an aggregate every time, or calculate aggregates for neighboring, potentially overlapping, regions. Such analyses can greatly benefit from query-driven materialization approaches that store and reuse intermediate or even full query results.

Naturally, in classical OLAP settings, query-driven materialization and result recycling are widely used and well understood [24, 35, 42, 45]. However, these methods do not address multi-dimensional spatial data. While methods have also been proposed for spatio-temporal OLAP queries, such as nanocubes [21] and the aR-tree [30, 31], these do not provide *precision guarantees* for spatial aggregation queries over *arbitrary polygons*. Both nanocubes and the aR-tree store aggregate information in a hierarchy of rectangles, maintained using a quadtree and an R-tree, respectively. Therefore, they are designed for aggregate queries over rectangular regions while their precision depends on the granularity of the underlying index structure. Using them to compute aggregates over *polygonal* regions introduces an approximation error, which *cannot be bounded*. There are also some analysis tools, such as Uber Movement [53], that rely on precomputation to provide exact results for spatial aggregations over polygons. However, they require the polygonal regions to be pre-defined at aggregation time. This assumes a priori knowledge of the workload and is thus not applicable in *exploratory* analyses, where the query polygons are chosen *ad-hoc*.

We propose GeoBlocks, a novel pre-aggregating data structure for geospatial point data that guarantees error-bounded results for spatial aggregation queries over arbitrarily shaped polygons. Essentially, GeoBlocks are materialized views on geospatial point data that pre-compute filters and aggregations on pre-defined

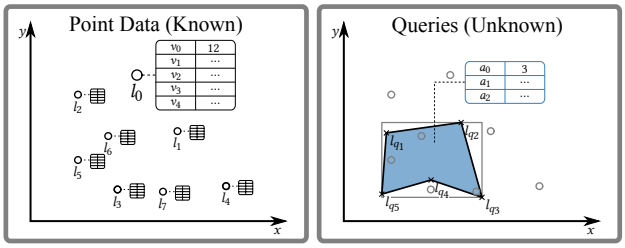


Figure 2: Problem overview: Calculating unknown aggregates a from known points P contained within an unknown query polygon R (specified by its vertices l_q).

columns. Instead of pre-computing aggregates over a hierarchy of rectangles as in prior work, GeoBlocks pre-compute aggregates over *fine-grained grid cells*. As depicted in Figure 1, GeoBlocks subdivide the spatial domain into grid cells, keeping aggregates for each individual cell. We allow the user to specify the geospatial granularity, and thereby *bound* the spatial approximation error. In addition, we propose a trie-like data structure that caches aggregates for commonly queried regions in a compact manner, enabling even faster response times. GeoBlocks are designed for historical point data and are thus write-once/read-only. However, while GeoBlocks currently do not support updates, they can be adapted to do so, as we briefly discuss in Section 5. Our contributions are summarized as follows:

- We propose GeoBlocks, the first, to the best of our knowledge, data structure that supports spatial aggregation over arbitrary polygons, while guaranteeing a bounded error.
- We develop a query-driven caching mechanism that further accelerates aggregate queries by leveraging the skew commonly found in exploratory query workloads.

The advantages of our approach are amply clear from our extensive experimental evaluation on real-world data. The results show that GeoBlocks achieve up to three orders of magnitude speedup compared to on-the-fly aggregation approaches and support sub-second response times.

In the remainder of this paper, we first formalize the problem in Section 2. Section 3 describes our approach, which we then experimentally evaluate in Section 4. Section 5 summarizes the key points discovered in the evaluation and discusses updates for GeoBlocks. Finally, we present an overview of related work in Section 6 before concluding in Section 7.

2 PROBLEM STATEMENT

In this paper, we propose a new data structure to speed up the execution of spatial aggregation queries. Formally, the query can be defined in SQL-like notation as follows:

```
SELECT AGG(P.v0), . . . , AGG(P.vk) FROM P
WHERE P.l INSIDE R(lq1, lq2, . . . , lqm) [AND filterCondition]*
```

Given a set of annotated points of the form P($l, v_0, v_1, \ldots, v_n$), where $l = (x_l, y_l)$ is the location of the point and v_i are numerical or temporal attributes, this query extracts multiple aggregates $a_i = AGG(v_i)$ over all the points contained in a query region R. The query region can be any *arbitrary polygon*, and its geometry is defined by the locations of the polygon's vertices $l_{q1}, l_{q2}, \ldots, l_{qm}$. The aggregates are non-holistic functions such as count, sum, min, max, or average. Finally, the query can have zero or more `filterConditions` on the attributes.

cell	Rectangular area, hierarchically subdividable into four children
cell level	Number of subdivisions performed on the spatial domain to obtain the cell
cell id/spatial key	Unique one-dimensional identifier of a cell
block level	Level of grid cells in a GeoBlock
cell aggregate	Aggregates of all tuples of a grid cell
cell covering	Error-bounded approximation of a polygon using cells

Table 1: Terminology

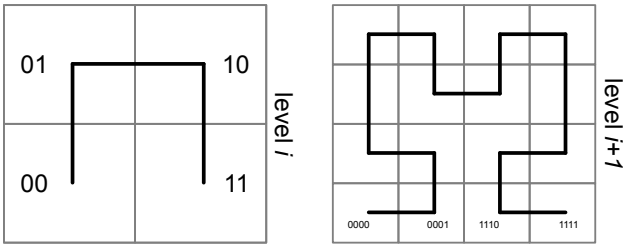


Figure 3: Hierarchical cell decomposition [16].

In exploratory interactive analyses, users can dynamically and unpredictably change not only the filtering conditions and the requested aggregates but also the polygonal query region. The data points, on the other hand, are known a priori. Figure 2 presents an example of this scenario: The left-hand side shows the input points that are located at $(l_0, \ldots, l_7)$ and have five attributes each. The right-hand side shows the query; the polygonal region is marked in blue, while three different aggregates are extracted. As can be seen in the figure, this query applies the aggregation over the three points that are contained in the query region, located at l_5, l_6, and l_7.

Existing approaches for spatial aggregation queries, such as the aR-tree [30, 31], are designed for rectangular regions, and thus *do not support arbitrary polygons*. Applying them to the example of Figure 2 requires to approximate the query polygon with a minimum bounding rectangle, displayed in grey, over which the aggregation is performed. This introduces an extra point in the results, l_3, which is outside the actual query region.

3 GEOBLOCKS

In this section, we first present the geospatial decomposition that forms the basis of our approach. We then discuss how we can quantify and *bound* the error that this decomposition introduces. Next, we explain the core concepts of GeoBlocks, their storage layout, and the efficient evaluation of spatial aggregation queries using GeoBlocks. Finally, Section 3.6 outlines our query-driven caching mechanism that further improves performance by leveraging the characteristics of the query workload. Table 1 provides an overview of the concepts introduced in this section.

3.1 Geospatial Decomposition

GeoBlocks rely on a hierarchical, quadtree-based spatial decomposition. In this decomposition, a given area (cf. the outer rectangle in Figure 3) is recursively subdivided into equally-sized smaller areas that we call cells. Each cell has four children, which

leads to an exponentially growing number of 4^n cells after recursively subdividing a cell n times. We encode each subdivision using two bits, which allows us to uniquely identify a cell at level n by concatenating the encoding of levels 0 to n. Equivalently, all cells at a given level can be enumerated using an *order-preserving space-filling curve*. Since children cells share a common prefix with their parent cell, containment tests are reduced to efficient bitwise operations. This encoding further allows storing cell ids in prefix-encoded index structures such as radix trees [16, 17] or in learned indices [52] to speed up containment queries. Figure 3 shows the decomposition of a cell in four (level i) and 16 (level $i+1$) sub-cells, and the corresponding enumeration with a Hilbert curve. Applying our decomposition strategy to the Earth's surface, we only need 64 bits to address every single square centimeter. That way, we map two-dimensional geospatial locations (lat/long coordinates) to one-dimensional 64-bit keys. In our implementation, we use the Google S2 library [38] to perform the spatial decomposition and cell enumeration. Note, however, that our approach is not restricted to S2 or the Hilbert curve. Any other framework that supports recursive geospatial subdivisions and order-preserving cell enumerations can be used instead.

Point Approximation. We map locations (i.e., *points*) to the smallest cell that contains them. The imprecision introduced by this approximation (e.g., at most 6.1 mm for any point in the US) is negligible, as the imprecision of GPS data is often orders of magnitude worse [54].

Polygon Approximation. Similarly, we approximate the query polygons on-the-fly by mapping them to a set of cells, possibly at different levels, as shown in Figure 4 (center and right). We call this geometric approximation a *cell covering*. In our implementation, we calculate cell coverings using the S2 library.

3.2 Bounded Error

Similarly to all geometric approximations, our cell covering introduces a spatial error. This is because all the cells that intersect the polygon outline, even minimally, are considered to be part of the polygon. However, in contrast to other coverings like the widely used minimum bounding rectangle (MBR), our cell covering is much more fine-grained. As can be seen in Figure 4, the cell covering approximates the polygon outline much more closely compared to the MBR. More importantly, the introduced approximation error can be bounded. In fact, any point on the cell covering is within a distance $\sqrt{\epsilon_1^2 + \epsilon_2^2}$ from the polygon outline, where ϵ_1, ϵ_2 are the side lengths of the cell. Clearly, the smaller the cell size, the smaller the approximation error. Consequently, our cell covering can *guarantee* a user-defined error bound, i.e., a bound on the spatial distance between the approximate and the original polygon, by using an appropriately small cell size. The MBR cannot guarantee such a bound, because its spatial extent, and thus its distance from the polygon outline, depends on the polygon's minimum and maximum coordinates in each dimension and cannot be controlled [52]. The user can specify the error bound by choosing an appropriate cell level[1] so that the cell's diagonal is not greater than her desired error. This user-controlled and bounded spatial error is the only error in GeoBlocks. All further operations are exact and do not introduce any additional error. While the error bound should be the driving factor when selecting a cell level, there are other points to consider: (1) The cell diagonal is the maximum error, and the average error can be expected to be lower. (2) The cost of reducing the error is not linear. Per each level, the diagonal, and thereby the error bound, reduces by a factor of 2. At the same time, the number of grid cells, and thus the query input, grows by a factor of 4.

[1]From the table at https://s2geometry.io/resources/s2cell_statistics

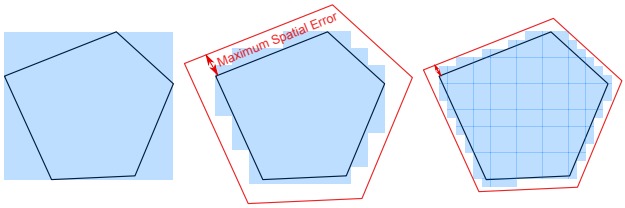


Figure 4: MBR (left) and two cell coverings with increasingly fine-grained resolution.

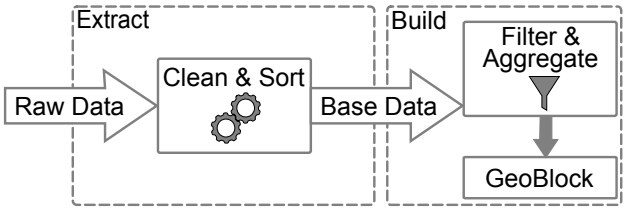


Figure 5: Creation of a GeoBlock in two phases. The extract phase is run once per dataset. The build phase is run for each filter and error bound combination.

3.3 Preprocessing

In addition to transforming the two-dimensional input space to one-dimensional spatial keys, we perform some additional preprocessing steps on the known point data. Our process, outlined in Figure 5, consists of two phases, extract and build, and is similar to the ETL process traditionally applied in OLAP settings. In the first phase, we prepare the raw data by filtering outliers in the often dirty datasets and limiting the columns to those relevant and suitable for analysis. We furthermore sort the data by the generated one-dimensional spatial key. This extract phase is run exactly once per dataset and allows us to cheaply build GeoBlocks from the extracted base data. The second phase, build, utilizes the clean and sorted base data to generate a GeoBlock in a single pass and thus in linear time.

Updates and Filters. An important part of data analysis is filtering to gain insights into the desired subsets of the data. In our process, we could apply filters either before or after sorting the raw data. While the first option seems tempting, as it would reduce the number of tuples over which the expensive sorting has to be performed, we decided to filter the data in the build phase. This way, we can utilize the sorted base data to quickly build GeoBlocks for different filter predicates, aggregates, and grid resolutions in a single pass. Building new GeoBlocks quickly is especially useful in exploratory analyses, where the data and filters of interest might not be fully known a priori. However, the increased cost of sorting all data has to be amortized over multiple GeoBlocks and filter predicates. In reality, the sorting cost might be amortized immediately, as some exploratory queries might need to compare a subset of the data with the total. Consider, for example, a query comparing the tip rate of expensive taxi rides (`WHERE fare_amount > 20`) with that of all rides. In this case, we would need to build a GeoBlock for all rides, and therefore sort the entire dataset anyway.

Given k different filter predicates with average selectivity s and a total input size of n tuples, we can calculate the runtime of building isolated GeoBlocks with filters before sorting, and incremental builds from sorted base data as follows:

$$k * (O(n) + O(sn * log(sn)) + O(sn)) \quad (1)$$

$$O(n * log(n)) + k * O(n) \quad (2)$$

The isolated build (1) has three phases, cleaning and filtering in $O(n)$, sorting in $O(sn * log(sn))$, and finally aggregating in $O(sn)$. Incremental builds (2) have a fixed component composed of cleaning and sorting in $O(n * log(n))$, followed by the incremental filtering and aggregation of the GeoBlock in $O(n)$. For incremental builds to pay off, the sorting cost of the regular builds ($k * (O(sn * log(sn)))$) has to outweigh the initial cost of the incremental builds ($O(n * log(n))$). As we only have runtime classes for each variant and de-facto runtimes will vary between systems and datasets, we cannot determine when amortization is reached solely depending on k and s. However, we provide an in-depth experimental analysis of the amortization in Section 4.

3.4 Storage Layout

Once the filtering of the base data is completed, we can start aggregating and building a GeoBlock. To build a GeoBlock, for each grid cell in the decomposed space, we compute a number of aggregates over all the tuples that it contains. Empty cells that do not contain any tuples are omitted during aggregation as they would needlessly consume space. We refer to the aggregates of a grid cell as *cell aggregates*. A GeoBlock stores cell aggregates in ascending order of the cell's spatial key, which is the same sorting order as the one applied to the base data. Moreover, a GeoBlock maintains a global header that combines all cell aggregates into a single GeoBlock-wide aggregate and contains additional metadata required for querying, such as the minimum and maximum cell id in the GeoBlock.

Cell Aggregate. Each cell aggregate stores pre-computed answers for spatial aggregation queries at the grid cell level. A cell aggregate consists of the cell's spatial key, the base data offset of the first tuple contained in the cell, and the number of contained tuples. Furthermore, it maintains aggregates for all columns (both numeric and temporal attributes) in the extracted data. The maintained aggregates are the minimum, maximum, and sum of all values contained in the cell. Note that using the sum together with the tuple count allows us to also compute the average as sum/count. Furthermore, the cell aggregate stores the minimum and maximum keys of the spatial column. The table in Figure 1 shows an example of a cell aggregate.

Aggregate Granularity. As described in Section 3.2, the block level (i.e., the granularity of the space decomposition) is defined by the user at build time. However, it is also possible to adapt the granularity at a later time. Building a more coarse-grained GeoBlock from an existing one is rather straightforward and does not require re-scanning the base data. We can easily combine all cell aggregates of the finer-grained GeoBlock corresponding to a more coarse-grained grid cell in a single pass over the aggregates. On the other hand, building a more fine-grained GeoBlock requires scanning and further subdividing the base data.

3.5 Querying

GeoBlocks support two variants of spatial aggregation queries. On the one hand, they support regular SQL `SELECT` queries that take a query polygon and produce a user-defined subset of the available aggregates. On the other hand, they support a specialized efficient implementation of `COUNT` queries that only report the number of points contained in a query polygon. Such `COUNT` queries are commonly used in analytics, especially in the context of visualization. Figure 1 shows an example query that extracts a set of aggregates over the Lower East Side region, which is approximated by a cell covering (marked in blue). The answer is calculated by extracting and combining all the aggregates contained in the blue cells.

```
1  lastAgg = 0
2  def selectQuery(polygon):
3    queryCells = s2.coverPolygon(polygon)
4    # Prune search range
5    queryCells.pruneLess(globalHeader.minCell)
6    queryCells.pruneGreater(globalHeader.maxCell)
7
8    lastAgg = 0
9    resultAggregates = initial
10   for qcell in queryCells:
11     # Map qCell to smaller childCells at the block level
12     childCells = s2.childrenAtLvl(qcell, BLOCK_LVL)
13     for cell in childCells:
14         getAggregates(cell, resultAggregates)
15   return result
16
17 def getAggregates(cell, result):
18   # Check the last results successor
19   if lastAgg == 0:
20     # Search initial header
21     aggregate = allAggregates.upperBound(cell).prev
22     if aggregate.cell == cell:
23       combineAggergates(aggregate, result)
24     lastAgg = aggregate
25   else:
26     if lastAgg.next.cell == cell:
27       lastAgg = lastAgg.next
28       combineAggregates(lastAgg, result)
```

Listing 1: SELECT query

To answer a spatial aggregation query over a polygonal region (Figure 6a), the polygon is approximated using a cell covering, as discussed in Section 3.1. We compute a cell covering that conforms to the error bound (Figure 6b). Note that the cell covering can have cells at different levels, and some of them might be larger than our grid cells. Such larger cells can be easily mapped to smaller grid cells (Figure 6c) in the GeoBlock and offer further optimization potential, as discussed next. The cell covering, however, cannot contain any cells smaller than the cells of the GeoBlock. Once we obtain the cell covering, we query the GeoBlock for each of the covering cells, as visualized for a `SELECT` query in Figure 6d. We then combine these partial results to compute the final result for the entire query polygon. In the following, we describe the query process for each cell of the covering. First, we use the GeoBlock's header to check if the cell overlaps with the GeoBlock at all. Thanks to the prefix-based containment checks, this is possible in constant time using the minimum and maximum cell id in the GeoBlock. Only if there is a possible overlap, we continue with the specific checks for `SELECT` and `COUNT` queries as follows:

`SELECT` Queries. `SELECT` queries have to look at all cell aggregates contained in the query cell. Listing 1 presents the pseudocode of the algorithm. After a query cell has passed the first check, we try to further limit the search space to the overlapping area (Lines 5 & 6). After splitting the query cell to smaller cells that match the GeoBlock's granularity if needed (Line 12), we locate the first intersecting grid cell using an upper-bound binary search (Lines 21 - 24). For all the following cells, we exploit the

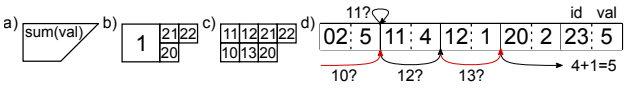


Figure 6: Query overview: Query polygon (a), cell covering (b), grid-cell representation of covering (c), and subquery for covering cell *1* in the cell aggregates (d, Listing 1 Line 12 and following).

```
1  def countQuery(polygon):
2    queryCells = s2.coverPolygon(polygon)
3    result = 0
4    for c in queryCells:
5      f_child = c.firstChildAtLvl(cell, BLOCK_LVL)
6      l_child = c.lastChildAtLvl(cell, BLOCK_LVL)
7      # Get first & last contained aggregate
8      first = allAggregates.lowerBound(f_child)
9      last = allAggregates.upperBound(l_child, first)
10
11     cnt = last.offset + last.count - first.offset
12     result += cnt
13   return result
```

Listing 2: `COUNT` query

fact that cell aggregates are stored contiguously in ascending order. This allows us to iterate over the cell aggregates (Lines 25 - 28) until we reach a grid cell not contained in the query cell, combining all cell aggregates along the way into the query result.

`COUNT` Queries. Intuitively, we can answer `COUNT` queries faster than `SELECT` queries, as we can exploit the sorted order of the cell aggregates to calculate the count without accessing the cell aggregates of all grid cells that are contained in the query cell. Specifically, `COUNT` queries can be answered using the count and offset values of only the first and the last cell aggregates that are contained in the query cell, as outlined in Listing 2. Note that here we benefit from having larger query cells. The larger the cells used in the covering, the fewer cell aggregates we need to access overall. To find the first and last cell aggregates, we calculate the id of the first and last child of the query cell at our grid level. We then locate the first child in the aggregates using a lower bound binary search (Line 8). Then, we use the position of the first child as a search start to locate the last child, again with a binary search (Line 9). Once we have located the aggregates of the first and last contained child, we can calculate (Line 11) the resulting count in a range-sum manner as:

$$\text{child}_{\text{last}}.\mathit{offset} + \text{child}_{\text{last}}.\mathit{count} - \text{child}_{\text{first}}.\mathit{offset}$$

3.6 Query-Cache Acceleration

While our cell aggregates can speedup queries significantly, there is further potential in pre-computing aggregates for frequently queried areas. This is based on the following key observations:

(1) Exploratory analyses are often repetitive in nature. Analysts, e.g., may run consecutive queries for the same area to extract different aggregates (i.e., using a different aggregate function, or aggregating over a different attribute).
(2) Furthermore, analysts might only iteratively change the shape or size of the query polygon. Consequently, part of the polygon's interior area remains unchanged.
(3) Lastly, analytical queries often focus on a geographic subset of the whole data. For the analysis of the NYC taxi data, e.g., the focus lies mostly on Manhattan, Brooklyn, and the airport regions, ignoring most suburbs [40].

In all the above cases, it is reasonable to pre-aggregate small grid cells that are often queried together to avoid costly scans of individual cells. In our example in Figure 1, e.g., we want to keep a single aggregate for the purple region, instead of having to consult all 64 contained cell aggregates.

Determining Relevant Aggregates. We want to determine the relevant areas that are worth being additionally pre-aggregated and cached, without making any assumptions about the expected query workload or the semantics of the indexed data. To achieve that, we use all previously seen queries as hints. Precisely, to determine whether an area is worth aggregating, we consider (i) the number of times it was queried, and (ii) its cell level.

For each query cell that intersects with the GeoBlock, we keep track of the number of times it was queried in a trie-like structure. We then use these statistics to calculate cell scores. The score of a cell is the sum of the cell's hits and the hits of its parent. This score takes into account that child cells can be used to speed up queries for parent cells. We then sort all cells by descending score. When scores are identical, we sort by ascending level (coarser-grained cells come first). As the last criterion, to ensure determinism, we sort by spatial key. We chose the above metric as it is sufficient to properly and repeatably represent the skew in the experiments in our evaluation while being easy to understand and implement. However, we also identified some weaknesses of our metric:

- Smaller cells might overshadow slightly less frequently queried bigger cells. Consider, for example, the green and purple cells of Figure 1 and assume that the green cell is queried just once more than the purple one. Based on our metric, we would then aggregate the green cell even if the purple cell could have an up to 64× bigger impact.
- The parent-child relationship is simplified: Children only cover parts of their parent but are treated as equally useful. Furthermore, we do not consider calculating aggregates by combining the aggregates of the parent and siblings of a cell. For example, the count for a cell could be calculated by subtracting the count of its sibling cells from the count of its parent cell.

Our evaluation showed that these shortcomings have a minor impact, but we plan to investigate them further and address them, if needed, in our future work.

Aggregate Storage. We cache aggregates in a trie-like cache, which we call AggregateTrie. Further, we allow the user to control the maximum size of the storage available for caching, and we store the AggregateTrie in-place with our cell aggregates and the filtered base data. As the cells are strictly ordered, we can simply insert the most relevant unaggregated cell until the reserved area is filled. Figure 7 shows an example AggregateTrie.

The storage for the aggregates is split into two parts. The first part (up until `0x90`) contains the trie structure, while the second part stores the actual aggregates. The root of the trie corresponds to the cell level that can enclose our input data, which is typically just a small fraction of the possible earth-wide input space. Each following trie-level encodes exactly one cell level, resulting in a fanout of 4. Since we store the AggregateTrie in-place, we chose a compact encoding storing all nodes contiguously. Nodes consist of just two 32-bit integers. The first one is the pointer to the first child in the AggregateTrie. The second one is the pointer to the corresponding aggregate in the aggregate storage (e.g., `0xb8`). Pointers are encoded as 32-bit offsets from the start of the allocated memory region. Both aggregates and nodes can be sufficiently encoded with an offset, as they are of fixed size.

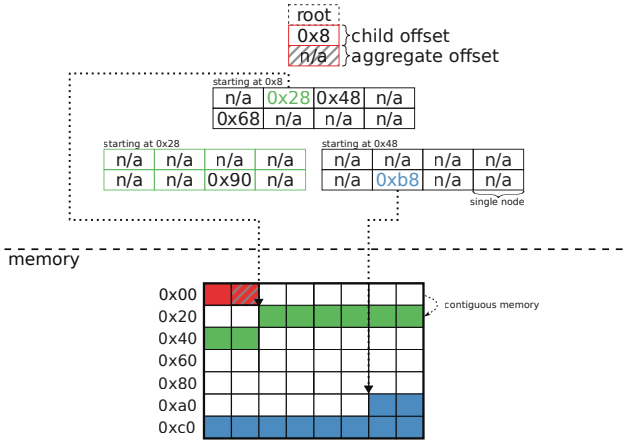


Figure 7: AggregateTrie with 40 byte aggregates and in-memory representation. Non-existent children (or aggregates) are marked with *n/a* and are encoded as `0x0`.

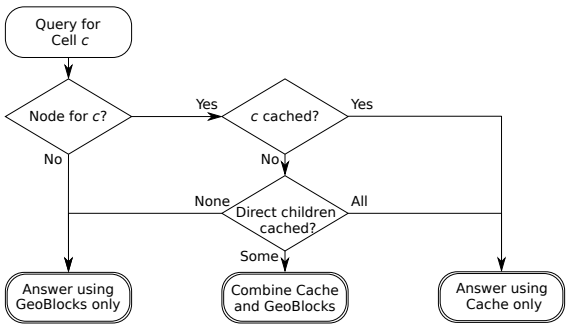


Figure 8: Overview of adapted query algorithm.

Nodes occupy 8 bytes, while the size of the aggregates depends on the schema. Since we store only the offset to the first child, we need to always allocate space for all children in a node, even for children that do not exist in the cache. This can be seen for the node starting at `0x28`, where only one child has an aggregate and no other children or aggregates exist. While this seems wasteful at first, the alternative would be to store four individual child offsets per node. As children are only created and stored if they are needed, our encoding never occupies more storage than this alternative. In fact, our design is more space-efficient in all cases, except for this worst-case in the example above, where only one out of four children is required.

Adapted Query Algorithm. We integrate the cached aggregates into the query algorithm (cf. Section 3.5). As the runtime of `COUNT` queries is mostly independent of the cell level since only the first and last grid cells are relevant, we do not expect noticeable speedups for them. Therefore, the adapted process, highlighted in Figure 8, is only used for `SELECT` queries.

Once the pre-query checks are completed, we first probe the query cache and resort to the old algorithm only when necessary. For each query cell, we traverse the AggregateTrie to locate the corresponding node. If there is no node for this cell, we abort probing and answer the query with the old algorithm. Once the node corresponding to the cell is reached, there are two possible ways forward. If the cell is cached, i.e., if it has a valid aggregate offset, the aggregate is extracted as a result. If the cell is not cached, there has to be at least one child at any level residing in our cache, as nodes are only created on demand. While, theoretically, all children could be used to reduce the number of grid cells of the GeoBlock to query, the number drops with each level, while keeping track of the missing children gets increasingly expensive. Therefore, we only consider direct children for this optimization. If some of the direct children are cached, we combine their aggregates with the results of the old algorithm for the non-aggregated ones to obtain the final result.

4 EXPERIMENTAL EVALUATION

We compare GeoBlocks with on-the-fly aggregation approaches on real-world data. To show that our advantage is not dependent on the indexing strategy, we use different strategies to index the base data of the on-the-fly approaches. We also compare GeoBlocks against a pre-aggregating approach, the aR-tree [30, 31]. However, we do not include the aR-tree in all experiments, as it is designed for rectangular queries and does not directly support polygonal ones.

4.1 Experimental Setup

Baselines. To keep the experiments as fair as possible, we use the mapping from geospatial space to linear space for the baselines as an index key unless specified otherwise. Furthermore, we keep all data in a columnar layout. Below, we describe the three strategies that we use to index the raw data, as well as our pre-aggregating baseline:

BinarySearch: This is the simplest baseline. Instead of indexing the data, we use binary search to locate the first and last contained raw tuple in the data. Afterward, we loop over all tuples in between and compute the requested aggregates. GeoBlocks use binary search to locate the cell aggregate in a similar way.

BTree: We use the BTree as a secondary index over the raw data. For the experiments, we use an open-source B-tree implementation by Google [7]. We probe the tree for the first child and scan the sorted raw data until no further tuple qualifies.[2]

PHTree: Our last non-aggregating baseline is a multidimensional point index structure, the PH-tree [56]. Instead of the one-dimensional spatial key, we use the latitude and longitude of the points to index the data. As the PH-tree only supports rectangular range queries, we use S2 to get the interior rectangle of the query polygon and use this as a query region. This way, we hope to keep the comparison fair, if not favorable for the PHTree, as this interior rectangle covers fewer points than our approach. As a consequence, the PHTree's query results differ from the results of the other approaches. For the measurements, we use an open-source C++ implementation [36].

aRTree: We implement the aR-tree [30, 31] based on the boost R-tree [5]. To minimize overlaps between nodes and thereby optimize the query performance, we use the R^* algorithm. In our implementation, each node covers a region r and has up to 16 child nodes, which further subdivide r into smaller areas. For each node, we store the aggregates in a cell aggregate corresponding to the region covered by the node, and reference it with an offset (cf. Figure 9). That way, we can modify the RTree query logic by adding *early abortion* exactly like in the aR-tree. Given a search area s and a node of the aR-tree that covers a region r, we distinguish three cases, as shown in Listing 3: (a) If s is completely contained by the covered region r_c of one of n's

[2]We first tried the PointIndex of the S2 library (https://s2geometry.io/devguide/cpp/quickstart.html#s2pointindex) that uses the same b-tree as point storage. Initial measurements showed that our optimized BTree implementation outperformed the PointIndex by 3×, so we opted for our implementation.

```
def queryARTree(node, searchArea, result):
  partiallyOverlappingNodes = []

  for child in node:
    if child.contains(searchArea):
      return queryARTree(child, searchArea, result)
    if searchArea.contains(child):
      result += child.aggregatedResult
    else if searchArea.intersects(child):
      partiallyOverlappingNodes.append(child)

  for child in partiallyOverlappingNodes:
    result += queryARTree(child, searchArea, result)
  return result
```

Listing 3: aR-tree lookup query

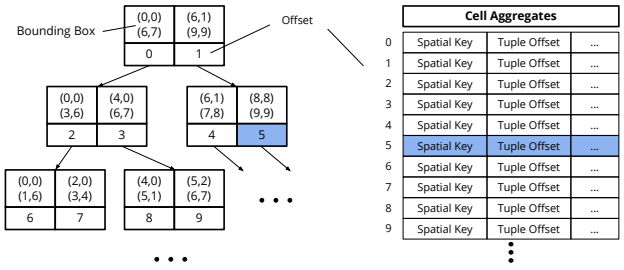


Figure 9: Illustration of aRTree with node size two and offsets into the cell aggregates.

child nodes, we recursively continue the search at the child node and do not consider other overlapping child nodes as this would result in counting values multiple times. (b) If the region covered by a child node is completely contained within the search area, we add its aggregated value to the overall result and continue processing the next child node. (c) If s and the child node region intersect, we *mark* the child node to be processed later *iff* no other child node fulfills criterion (a).

By accepting that points are counted multiple times in the case of overlapping internal nodes, our aR-tree implementation follows the query algorithm of the original aR-tree that does not consider overlapping children. While the implementation delivers an upper-bound of the result, it visits the internal nodes in the same way the aR-tree does, thus achieving the same performance.

Implementation. We implement GeoBlocks in C++ as described in Section 3. Our implementation, as well as that of all baselines, is single-threaded. Throughout this section, especially in all figures, we will refer to GeoBlocks as Block. Furthermore, we will differentiate between the regular Block and Block^{QC}. Block denotes GeoBlocks without query caching using the basic query algorithm. Block^{QC} is GeoBlocks using query caching with the AggregateTrie and adapted query process outlined in Figure 8.

Hardware. All experiments are run on a server machine with two Intel Xeon E5-2680 v4 processors clocked at 2.4 GHz. The machine is equipped with 256 GiB of DDR4-2400 RAM. All performed experiments fit entirely into main memory.

Dataset. The primary dataset used in the experiments is composed of trip records from 12 million NYC yellow cab rides in the time between January and March 2015, which we cleaned of outliers. It is openly available for download from the NYC Taxi and Limousine Commission (TLC) [49]. It contains data from individual rides like pickup and drop-off location and time, passenger count as well as trip distance.

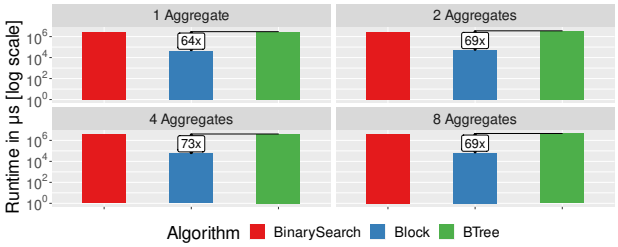


Figure 10: Runtime with increasing number of aggregates.

Unless otherwise specified, the queries consist of polygons representing NYC neighborhoods taken from [25]. As a base workload, we build a query containing each polygon once. For the skewed workload, we select 10% of neighborhoods uniformly at random and query them multiple times. We select 7 aggregates, requesting each column at least once, as query output.

In addition, we use 8 million geotagged tweets from the contiguous US and query them using polygons representing US states. Finally, we use an extract of 389 million OpenStreetMap (OSM) points in the Americas and query them with polygons representing countries. Both these datasets have randomly generated integer values as payload. For both, we fix the level at 11 (~7km diagonal). Unless otherwise specified, all experiments are conducted on the primary dataset only.

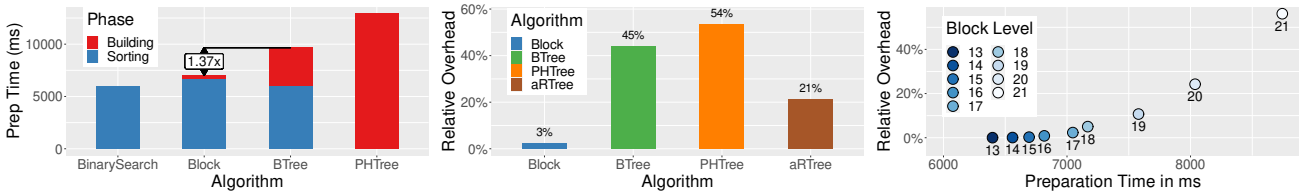
4.2 Baseline Comparison

Impact of Number of Aggregates. To show the impact of the number of aggregates on the performance of the baselines and the Blocks, we use a combined workload consisting of once the base and four times the skewed workload. We query this workload for 1, 2, 4, and 8 aggregates and report the results in Figure 10.

As one can easily see, GeoBlocks outperform both the BTree and BinarySearch baseline in all cases. We omitted the PHTree and aRTree from these experiments, as the imprecise rectangular approximation of the skewed workload lead to a drastic increase in their runtime. Even for the base workload, the PHTree was slower by a factor of about 3× while covering fewer tuples.

Indexing Overhead. We compare the build time, i.e., the preparation time required prior to running any query, in Figure 11a, with the block level set to 17 (~100m diagonal). The reported times for sorting are measured once for the optimized out-of-place sorting for the Blocks and reported for each baseline. This step is completely identical in all sorting baselines. There is a noticeable gap in the sorting phase between the BTree/BinarySearch and the Block. This gap is caused by the collection of grid cell ids to aggregate that we piggybacked on the sorting process to save an additional pass on the data. Overall, the Block is built faster than the BTree and the PHTree, and slightly slower than the BinarySearch, which only needs to sort the input data. We exclude the aRTree baseline from this experiment as we only optimized the implementation for query performance, and the build time was multiple orders of magnitude slower than the others described. Most notably, the majority of the Block preparation is spent on sorting, indicating that once the data is sorted, building additional Blocks with different filter sets is reasonably cheap.

The relative space overhead of each algorithm is depicted in Figure 11b. BinarySearch was omitted as it does not require any additional storage. One could argue that this is not a fair comparison to the BTree and PHTree as they index individual points, but



(a) Build time of GeoBlocks and baselines. **(b) Size overhead of GeoBlocks and baselines.** **(c) Level influence on GeoBlocks overhead.**

Figure 11: Index overhead in build time and space.

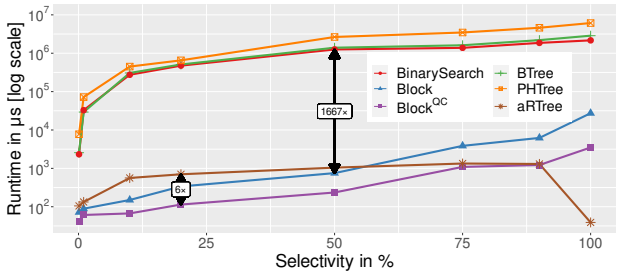
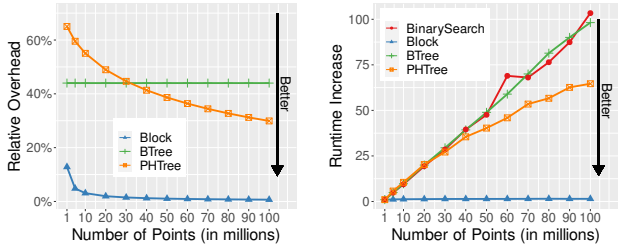


Figure 12: Query runtime for varying selectivity.



(a) Size overhead of GeoBlocks and baselines.

(b) Relative runtime increase of GeoBlocks and competitors compared to 1M points.

Figure 13: Scaling with increasing input sizes.

as our goal is to provide approximate results, we wanted to show that storing intermediate results is less space-consuming than one would assume for such fine-grained aggregates. While the aRTree is more space-saving when compared to the single-point indices, it still introduces an order of magnitude higher storage overhead than GeoBlocks.

Impact of Selectivity. Selectivity is usually defined based on a single query, but in our context, it is hard to specify what a single query is. We break down query polygons, e.g., the orange bordered Lower East Side in Figure 1, to different-sized cells covering the polygon (e.g., the purple cell), which in turn are broken down into equally sized cells to query (blue cells). While the intermediate cells of the query polygon's covering are the best representation of individual queries, as each index is probed once for them, they are artificial concepts introduced by our algorithm. Furthermore, these are hard to map to the rectangular query regions of the PHTree and the aRTree. Therefore, we define selectivity based on query polygons. For this experiment, we artificially select polygons covering a part of NYC, which contains a certain percentage of the total rides. Figure 12 reports the runtime of the base workload at different selectivities using a logarithmic scale. PHTree's and aRTree's measured selectivities differ slightly from the reported ones due to the less precise covering using an interior rectangle. As this covering contains fewer points, this should slightly skew the experiment in favor of the PHTree and aRTree. Even though GeoBlocks can handle rectangular queries as well, since rectangles are just constrained polygons, we opted for the most-precise covering where possible.

While runtime rises quickly for all baselines for selectivities above 1%, the increase is much softer for both Block variants. Even though the workload is not skewed, and we only use 2% of additional storage for query caching, Block^{QC} still outperforms the non-caching Block across all selectivities. This is likely explained by the shape of the polygons that are often simple quadrilaterals or pentagons. These can be covered using few cells and, therefore, most of these cells can be pre-aggregated. BinarySearch can keep up with the BTree, reporting similar runtimes independent of selectivity, while the PHTree lags behind quickly. Even if the relative runtime gap narrows for higher selectivity, the absolute gap still favors GeoBlocks. The aRTree, our implementation of the aR-tree, outperforms the on-the-fly aggregating benchmarks easily while staying behind GeoBlocks for lower selectivities. However, it can catch up with Block at around 50% selectivity. At 100% selectivity, the aRTree needs to only access the root aggregate, explaining the sharp drop in runtime. Overall, GeoBlocks outperform the non-aggregating baselines by at least two and up to three orders of magnitude, performing on-par with the aR-tree while delivering far more precise results.

Scalability. To study the performance for different-sized datasets, we collect 100M taxi rides spanning all of 2015 and build and query the approaches for an increasing subset of these rides. We omit the aRTree as the build time exceeded reasonable limits upward of 30 million points. As the build time is dominated by the sorting process, which is shared and identical in all approaches, they scale identically in build time. When comparing the size overhead in Figure 13a, we can see that the BTree overhead is constant as expected. For the PHTree, we see the positive impact of the integrated compression strategies for bigger datasets. Still, the near fixed-size grid aggregates - the size of a GeoBlock is determined by the spatial distribution of points, not their number - enables even smaller overheads for GeoBlocks. To focus on the individual scalability for queries, we analyze the query runtime normalized to the runtime of each approach for one million points. As shown in Figure 13b, both the BTree and the BinarySearch scale linearly with the input size, as the on-the-fly aggregation dominates the runtime. We expect a similar behavior from the PHTree, but as the covering is less accurate and chosen deliberately smaller, the increase is not fully linear. For GeoBlocks, the runtime stays nearly constant, since it depends on the number of maintained aggregates, and not on the number of individual points. The number of aggregates is in turn determined by the spatial distribution of the input. Since one million

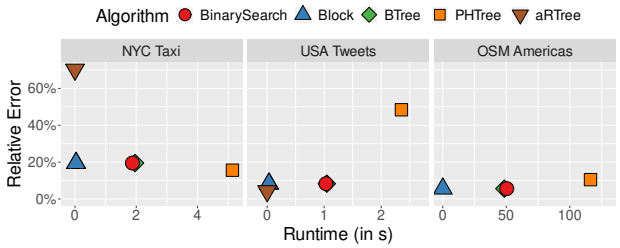


Figure 14: Query runtime and relative error for varying datasets.

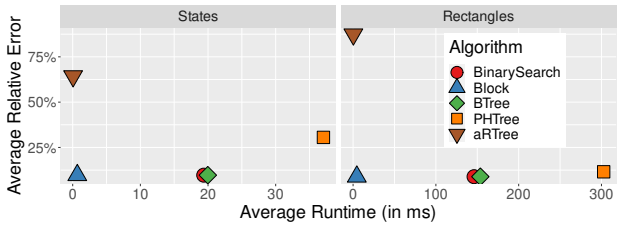


Figure 15: Query runtime and relative error for US states and generated rectangles on the Twitter dataset.

points already cover most areas in NYC, the distribution does not change when further increasing the number of points, i.e., the number of aggregates does not increase significantly. This explains why query latency remains nearly constant for bigger datasets.

Datasets. To show that our approach is not limited to the NYC taxi dataset, we evaluate it on the two additional datasets in Figure 14. We again query the whole area represented by the individual polygons and report runtime, as well as the average error defined as $\frac{|\text{\# tuples in query result} - \text{\# tuples in polygon}|}{\text{\# tuples in polygon}}$. For the OSM dataset, the aRTree again was excluded because of its excessive build time. As the Block, BinarySearch, and BTree use the same covering, the result and error are identical. While the aRtree and PHTree use an identical rectangular representation, the pre-aggregated nodes of the aRTree lead to a different result, and therefore error. Overall, the aRTree and Block are similarly fast with a slight advantage for the aRTree, outperforming the non-aggregating approaches easily. However, the error for Block is far more stable.

Accuracy. Finally, we want to study the influence of smaller individual polygons, as well as rectangular areas, on both runtime and relative error. Therefore, we query all US states and 51 randomly generated rectangles within the US on the Twitter dataset and report the average runtime and error in Figure 15. In contrast to the previous experiment, we query all areas individually. For both polygons and rectangles, the same overall trends are visible. The aRTree is slightly faster than Blocks as the large polygons can be answered in the upper levels of the tree. However, this leads to high imprecision even for rectangular queries as partially overlapping internal nodes might be counted multiple times. Besides, we see that the individual errors canceled out in Figure 14, leading to a seemingly good error bound. While the PHTree error also improves considerably for the rectangular workload, we expected it to be exact. We suspect this is caused by our transformation of the coordinates to integer space, which is necessary for efficient queries. As expected, the performance

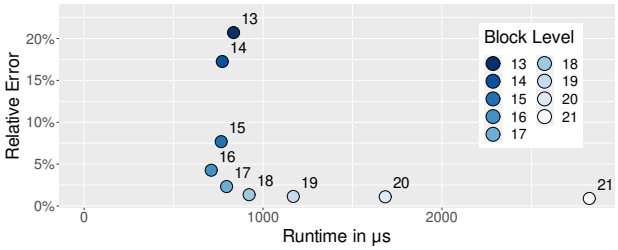


Figure 16: Relative error and runtime at varying levels.

Level	13	14	15	16	17	18	19	20	21
Sorting	6020	6008	6317	6459	6633	6754	7028	7344	7666
Building	376	499	376	356	411	408	538	666	1025

Table 2: Index build times in ms at varying levels.

of Blocks and the other approximating baselines does not degrade for rectangular areas. The aggregating approaches again far outperform the point indexing approaches in runtime.

4.3 Sensitivity Analysis

After showing that GeoBlocks easily outperform all baselines, we study the impact that the configuration of GeoBlocks has on throughput, as well as the impact of data skew on the adaptive Block version. The Block configuration is specified by three parameters: The first setting we study is the level of the Block, i.e., the resolution of the grid overlying the spatial domain. Next, we take a look at the impact of skew on both Block and Block^{QC}. Finally, we examine how the size of the AggregateTrie influences the runtime of unskewed and skewed workloads.

Impact of Block Level. We vary the block levels from 13 to 21 (between ~1.5km and ~6m diagonal) while keeping the other configuration parameters fixed. From a runtime-only point of view, lower-level (coarser-grained) blocks are always preferable, as the query algorithm needs to take fewer cells into account. However, this comes at the price of precision loss. Figure 16 illustrates the connection between the block level, the runtime, and the relative error introduced by the cell covering. The cell covering can introduce only false positive results, i.e., some reported results are not contained in the actual polygon. The figure clearly shows the expected overall trend: the higher the level, the lower the relative error and the higher the runtime. However, after a certain point, decreasing the level further does not pay off. Further, we see that the correlation between error and runtime is not linear, as we already suspected in Section 3.2. The correlation does not even follow the discussed influences completely, which is likely caused by missing sparse children, and the non-uniform distribution of points leading to a gap between the relative error and the configurable spatial error.

The block level influences not only the relative error and the runtime, but also the build time and size of GeoBlocks. Figure 11c depicts the build time and size overhead for GeoBlocks from levels 13 to 21. The build time seems to be only slightly affected by the level, rising slowly with it. Table 2 splits the runtime into two parts: sorting and building. There is a noticeable increase in sort time along with the block level, in addition to the expected increase in build time. This increase in sorting can be explained through our grid cell extraction that we piggybacked to the sorting process, which has to extract more finer-grained

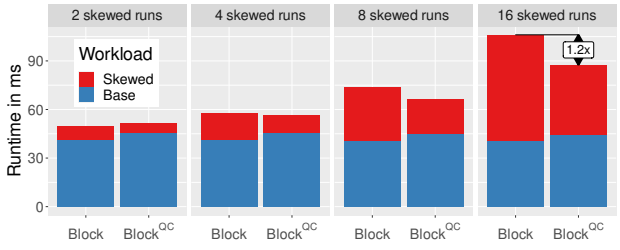


Figure 17: Query runtime with increasing workload skew.

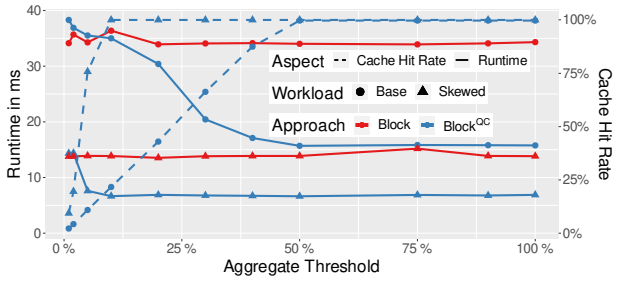


Figure 18: Impact of threshold on workload runtime (solid line) and cache hit rate (dashed line).

cells. The size overhead, however, grows exponentially due to the exponentially growing number of cells along with the level.

Impact of Skew. To study the impact of data skew on the effectiveness of query caching, we measure the query runtime when running the NYC workload once, and the skewed workload multiple times. The number of times we run the skewed workload varies in each experiment. We fix the block level to 17 (~100m diagonal) and the size of the cache to 5% of the cell aggregates, which roughly corresponds to aggregating all cells of the skewed workload. Figure 17 displays the absolute runtime for both the base and the skewed part of the workload. One can see that after four skewed runs, the cached aggregates start to pay off. With even more skew in the total workload, our query-caching Block^{QC} quickly starts to outperform Block. Furthermore, as expected, the runtime for the base workload stays nearly constant, and is always slightly faster for Block. This is easily explained by the overhead of probing the AggregateTrie for each cell, regardless of whether the cell is aggregated or not.

Impact of Aggregate Threshold. Having studied the impact of skew, we want to examine how the aggregate threshold, and thereby the size of the query cache (in Block^{QC}), influences the runtime of the base and the skewed workload. The aggregate threshold denotes the relative size overhead that the query cache, the AggregateTrie, introduces compared to the size of the cell aggregates in the regular GeoBlock. We again fix the block level to 17, and the number of skewed runs to four. Figure 18 depicts the measured runtimes and cache hit rates. The runtime of Block is unaffected by the changed threshold and only acts as a baseline to highlight the influence on Block^{QC}. Up until a threshold of around 5%, only queries from the skewed workload can be answered using the AggregateTrie. The small speedup in the base workload can be explained by the inclusion of the skewed workload in the base workload. Once all cells in the skewed workload are cached, and the cache hit rate for the skewed part reaches 100%, other query cells of the base workload start to get cached

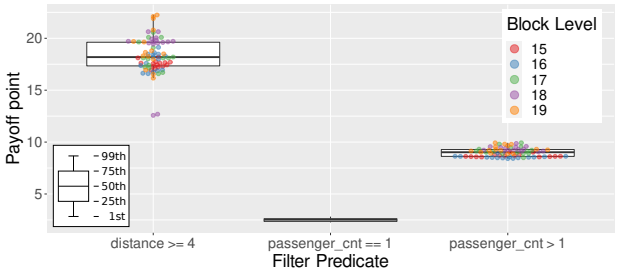


Figure 19: Payoff point: Number of incremental builds required to amortize the cost of sorting the raw data.

as well. While this, of course, leads to further runtime improvements, it is undesirable, especially when memory is scarce. In our experiments, at around 50%, the cache hit rate reaches 100% for both workloads, and there is no further speedup, even when the cache size is doubled. The cache hit rate, illustrated by the dashed line and shown on the right axis, shows the desired effect. The skewed part is cached almost immediately, and the hit rate for the unskewed workload grows linear with increasing cache size. The average lookup time slowly grows from 58ns at 1% to 81ns at 100%. As the lookup time depends on the number of levels (30 in the maximum) and not on the size, this growth is attributable to more complex access patterns for larger cache trees.

4.4 Changing Filters

Finally, we compare our process of Figure 5, wherein we build multiple GeoBlocks from the sorted base data, against building isolated GeoBlocks from scratch. We vary the block level from 15 to 19 (between ~420m and ~27m diagonal), and build 15 GeoBlocks per level using three different predicates of varying selectivity:

- **distance >= 4:** Long taxi trips, selectivity of ~16%
- **passenger_cnt == 1:** Solo taxi trips, selectivity of ~70%
- **passenger_cnt > 1:** Shared taxi trips, selectivity of ~30%

For this, we want to analyze how many different filter and level combinations are required to amortize the initial cost of sorting. Figure 19 shows the payoff point of filter changes for our three filter predicates. The payoff point is the number of incremental builds required to be, in sum, faster by creating incremental builds than building individual GeoBlocks from the raw data and filtering before sorting. We omitted the individual runtimes for the `passenger_cnt == 1` predicate as they would be too densely packed vertically.

As expected, the more selective the filter, the lower the speedup. Once all tuples in the raw data have been filtered according to the predicate, the qualifying tuples have to be sorted. More selective predicates take longer to amortize as sorting few tuples is cheap , whereas for the 70% selectivity query `passenger_cnt == 1`, the more expensive sorting is amortized almost immediately. There is a correlation between the block level and amortization, most notably for the most selective predicate `distance >= 4`. Given that the payoff point drastically rises with lower selectivity, we expect that incremental builds will only pay off when the new filters are less selective. If only a few highly selective queries are expected, building regular GeoBlocks directly from the raw data will still be the fastest option. However, the time to switch to a new filter, and therefore the individual query latency, will always be lower for incremental builds.

5 DISCUSSION

In this section, we discuss the takeaways of the evaluation as well as updates for GeoBlocks.

Evaluation Summary. First, we showed that pre-aggregation in a spatial context pays off when a limited and bounded spatial error is acceptable, independently of the number of aggregates queried and the selectivity of the query polygons. Furthermore, GeoBlocks can be built fast, introducing only a small overhead compared to the simple BinarySearch baseline. Even when the data is already indexed with one of our baselines (i.e., without taking the index build time into account), GeoBlock's build time of around 7 seconds can be amortized by fewer than 30 polygon queries with a selectivity of 10% (cf. Figures 11a and 12). In addition, building multiple GeoBlocks once the data is sorted is possible within one second for our dataset, cf. Figure 11a. Building new GeoBlocks for different filters is even faster when using sorted base data, often amortizing the initial extra cost of sorting all data in less than 10 filter changes (cf. Figure 19). Even though not all configurations are optimal for GeoBlocks, there are acceptable error-runtime trade-offs, in our case around levels 17 and 18. While the level does not significantly impact the index build time, the size overhead growth is almost exponential, cf. Figure 11c, indicating that it is wise to think about which error is acceptable for the given query workload when memory is scarce.

Updates. Up until now, we considered GeoBlocks to be read-only as they are designed for historical point data. However, the layout of GeoBlocks allows us to integrate updates easily, as long as a cell aggregate for the region of the newly arriving tuple already exists. For the non-adaptive version, all we have to do is locate the cell aggregate containing the tuple and update all stored aggregates. In the adaptive version, we additionally need to update all cached parents of the grid cell in the AggregateTrie as well. Thanks to the prefix-based indexing property of the trie, we can do this in a single depth-first traversal. Only if tuples arrive for a new, previously unaggregated region, we have to rebuild the aggregate layout, as we rely on the cell aggregates to be sorted. However, as we have shown in the evaluation, recalculating the cell aggregates is often possible within a second, so this operation would not induce too much delay when updates are implemented in batches instead of single tuples. Other indexing approaches on the cell aggregates (e.g., a clustered B-tree) could eliminate the need to rebuild by reserving storage for new aggregates. Preliminary experiments using `std::map` and a B-tree as an index showed similar lookup performance at the cost of increased size overhead.

6 RELATED WORK

Our approach builds on seminal work from decades of research on spatial indexing. Decomposing space into hierarchical grid cells [1, 9, 39], as well as approximating polygons using simpler shapes [18], are all well-known approaches. Likewise, enumerating cells using a space-filling curve such as Hilbert or Z order [26, 27] and storing aggregate information within cells [20, 30, 46] are ideas that have been around for some time. However, while building on these established concepts, GeoBlocks present the first pre-aggregating data structure that supports a *bounded, distance-based error* on the results of *polygonal queries*. Specifically, prior work on pre-aggregation [14, 30, 31, 34] is limited to rectangular queries and requires an expensive post-processing (refinement) step to answer polygonal queries. GeoBlocks, on the other hand, yield error-bounded results and do not require expensive refinement.

Spatial Aggregation. Past work has proposed several approaches for spatial aggregation queries [23]. These approaches mainly rely on pre-aggregation [14, 34]: they pre-aggregate records at various spatial resolutions and store this summarized information in a hierarchy of rectangular regions, maintained using a spatial index like the quadtree or the R-tree [19, 30–32]. For instance, the aRtree [30, 31] enhances the R-tree by storing aggregate information for each node. This allows to directly extract the aggregate of all the records contained in a node, if the node's MBR is fully enclosed in the query region. Being a variant of the R-tree, the aRtree constrains the supported queries to only rectangular regions. Furthermore, the computed aggregates are approximate and the error *cannot be bounded*, since the accuracy depends on the resolution of the rectangular R-tree nodes. Providing precision guarantees for arbitrary polygons requires accessing the raw data and involves additional processing. There are also approaches that store aggregates inside a data cube [6, 37], or using sketches [48]. Nanocubes [21], for example, store the CUBE operator for spatio-temporal datasets, and are specifically designed for visualization systems. The data cube-based approaches suffer from the same limitations as the aRtree, since they also rely on a hierarchy of rectangular regions. Besides, accessing the raw data to refine the aggregates might require additional indices, as the cube does not store individual records. Vorona et al. [55] approximate the distribution of geospatial points with an autoregressive deep learning model to answer arbitrary polygonal queries, but they cannot provide any error bounds. Pandey et al. [29] propose to use learned indices for query-efficient spatial indexing, albeit limited to range queries. Finally, Raster Join [51] uses GPU rendering to compute aggregates over a point-polygon join. In contrast, GeoBlocks support aggregation over spatial selections.

Prefix sums [10] can be used in addition to pre-aggregation to enable fast range-sums. This is achieved by only inspecting the aggregates in the two corners of a query region, rather than every aggregate inside the query region. An example of this is our `COUNT` algorithm. However, in contrast to our `SELECT` queries, these range-sums are unable to extract min and max aggregates.

Materialized Views and OLAP Cubes. GeoBlocks are essentially materialized views over geospatial data with support for filters and aggregations. In contrast to regular views [12, 44], GeoBlocks are designed for historical spatial data and can adapt to the query workload at a fine-grained level using a trie-like cache. Work on materialized view selection [2] also makes materialization decisions based on the query workload, but at a much coarser granularity (e.g., what columns to aggregate). There has also been a lot of work on data cubes and query caching [11, 15, 42], but these do not support geospatial data as a first-class citizen.

Spatial Point Indexing. Spatial point indexing approaches typically index points using a hierarchy of MBRs, most notably the R-tree [13], or by subdividing grid cells into equally-sized children, e.g., the quadtree [9, 39]. Both of these index structures are queried using the dimension-wise min/max values, i.e., the query regions are rectangular. Other approaches, like the UB-tree [3], assign univariate keys to the indexed regions first and rely on these keys for data access. While the UB-tree does not specify how these keys have to generated, most approaches use space-filling curves like the Z order [26, 27].

Based on these concepts, more specialized indices have been developed. The PH-tree [56] combines a quadtree with hypercubes to allow splitting all dimensions in each node, providing a space-efficient index structure for multidimensional data. The space efficiency can be partly attributed to the utilization of prefix

sharing, similar to the one used in our trie-like cache. Alternating the indexed dimensions in an in-memory tree structure, the BB-tree [47] offers fast point and range queries for multidimensional data. While these structures require the index to be built a priori, there are others like QUASII [33], where the index is built incrementally as a side product of query execution. As a result, QUASII can adapt to the query workload at runtime. However, QUASII only supports spatial range (window) queries. Recently, Shin et al. [43] proposed integrating grid indices into a tree structure to achieve faster node accesses and point operations.

7 CONCLUSIONS

We have introduced GeoBlocks, a novel pre-aggregating data structure for geospatial data. GeoBlocks pre-compute aggregates over fine-grained grid cells, thereby supporting arbitrarily shaped polygons. Using these aggregates, GeoBlocks can provide fast query results with a user-controlled spatial error. Furthermore, GeoBlocks can speed up aggregate queries for commonly queried regions by dynamically adapting to any given workload using limited additional storage.

Comparing GeoBlocks with on-the-fly aggregating indexing baselines, we have shown that we can outperform them for any number of aggregates, in parts by three orders of magnitude. The introduced storage overhead is comparable, and often even lower, to that of traditional indexing structures, while GeoBlocks can be built equally fast. Looking at GeoBlocks' configuration options, we have shown how they can be adapted to the given dataset and workload, and how they influence the runtime, the overhead, and the error in the result. Overall, GeoBlocks are materialized views over geospatial data that support filter predicates and aggregates while enabling fine-grained adaptation to the query workload.

REFERENCES

[1] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *PODS*, pages 265–272. ACM Press, 1990.
[2] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB*, pages 156–165, 1997.
[3] R. Bayer. The universal b-tree for multidimensional indexing: general concepts. In *WWCA*, pages 198–209. Springer, 1997.
[4] *How to analyse bike data for urban planning?* https://www.bikecitizens.net/analyse-bike-data-urban-planning/.
[5] *Boost R-tree.* https://www.boost.org/doc/libs/1_69_0/libs/geometry/doc/html/geometry/reference/spatial_indexes/boost__geometry__index__rtree.html.
[6] F. Braz, S. Orlando, R. Orsini, A. Raffaetà, A. Roncato, and C. Silvestri. Approximate aggregations in trajectory data warehouses. In *ICDE Workshops*, pages 536–545, 2007.
[7] Google code archive. https://code.google.com/archive/p/cpp-btree/.
[8] N. Ferreira, M. Lage, H. Doraiswamy, H. Vo, L. Wilson, H. Werner, M. Park, and C. Silva. Urbane: A 3d framework to support data driven decision making in urban development. In *Proc. IEEE VAST*, pages 97–104, 2015.
[9] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.
[10] S. Geffner, D. Agrawal, A. E. Abbadi, and T. R. Smith. Relative prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *ICDE*, pages 328–335, 1999.
[11] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.
[12] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, pages 98–112. Springer, 1997.
[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57. ACM Press, 1984.
[14] J. Han, N. Stefanovic, and K. Koperski. Selective materialization: An efficient method for spatial data cube construction. In *PAKDD*, pages 144–158. Springer, 1998.
[15] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216. ACM Press, 1996.
[16] A. Kipf, H. Lang, V. Pandey, R. A. Persa, C. Anneser, E. Tzirita Zacharatou, H. Doraiswamy, P. A. Boncz, T. Neumann, and A. Kemper. Adaptive main-memory indexing for high-performance point-polygon joins. In *EDBT*, pages 347–358, 2020.
[17] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. A. Boncz, T. Neumann, and A. Kemper. Approximate geospatial joins with precision guarantees. In *ICDE*, pages 1360–1363. IEEE Computer Society, 2018.
[18] H. Kriegel, H. Horn, and M. Schiwietz. The performance of object decomposition techniques for spatial query processing. In *SSD*, volume 525, pages 257–276. Springer, 1991.
[19] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, pages 401–412. ACM, 2001.
[20] I. Lazaridis and S. Mehrotra. Multi-resolution aggregate tree. In *Encyclopedia of GIS*, pages 764–765. Springer, 2008.
[21] L. D. Lins, J. T. Klosowski, and C. E. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2456–2465, 2013.
[22] Z. Liu and J. Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *Proc. TVCG*, 20(12):2122–2131, 2014.
[23] I. F. V. López, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: a survey. *IEEE Trans. Knowl. Data Eng.*, 17(2):271–286, 2005.
[24] F. Nagel, P. A. Boncz, and S. Viglas. Recycling in pipelined query evaluation. In *ICDE*, pages 338–349, 2013.
[25] NYC neighborhoods. https://data.cityofnewyork.us/City-Government/Neighborhood-Tabulation-Areas/cpf4-rkhq.
[26] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *SIGMOD Conference*, pages 326–336. ACM Press, 1986.
[27] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS*, pages 181–190. ACM, 1984.
[28] V. Pandey, A. Kipf, T. Neumann, and A. Kemper. How good are modern spatial analytics systems? *PVLDB*, 11(11):1661–1673, 2018.
[29] V. Pandey, A. van Renen, A. Kipf, J. Ding, I. Sabek, and A. Kemper. The case for learned spatial indexes. In *AIDB@VLDB*, 2020.
[30] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *SSTD*, pages 443–459. Springer, 2001.
[31] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *ICDE*, pages 166–175, 2002.
[32] D. Papadias, Y. Tao, J. Zhang, N. Mamoulis, Q. Shen, and J. Sun. Indexing and retrieval of historical aggregate information about moving objects. *IEEE Data Eng. Bull.*, 25(2):10–17, 2002.
[33] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki. QUASII: query-aware spatial incremental index. In *EDBT*, pages 325–336, 2018.
[34] T. B. Pedersen and N. Tryfona. Pre-aggregation in spatial data warehouses. In *SSTD*, pages 460–480. Springer, 2001.
[35] T. Phan and W. Li. Dynamic materialization of query views for data warehouse workloads. In *ICDE*, pages 436–445, 2008.
[36] *mcxme/phtree.* https://github.com/mcxme/phtree.
[37] F. Rao, L. Zhang, X. Yu, Y. Li, and Y. Chen. Spatial hierarchy and olap-favored search in spatial data warehouse. In *DOLAP*, pages 48–55. ACM, 2003.
[38] S2 geometry. https://s2geometry.io/.
[39] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.
[40] T. Schneider. *Analyzing 1.1 Billion NYC Taxi and Uber Trips, with a Vengeance.* https://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/.
[41] V. Shah. *Citi Bike 2017 Analysis - Towards Data Science.* https://towardsdatascience.com/citi-bike-2017-analysis-efd298e6c22c.
[42] J. Shim, P. Scheuermann, and R. Vingralek. Dynamic caching of query results for decision support systems. In *SSDBM*, pages 254–263, 1999.
[43] J. Shin, A. R. Mahmood, and W. G. Aref. An investigation of grid-enabled tree indexes for spatial query processing. In *SIGSPATIAL*, pages 169–178, 2019.
[44] O. Shmueli and A. Itai. Maintenance of views. In *SIGMOD*, pages 240–255. ACM Press, 1984.
[45] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB*, pages 488–499, 1998.
[46] S. Singla, A. Eldawy, R. Alghamdi, and M. F. Mokbel. Raptor: Large scale analysis of big raster and vector data. *PVLDB*, 12(12):1950–1953, 2019.
[47] S. Sprenger, P. Schäfer, and U. Leser. Bb-tree: A main-memory index structure for multidimensional range queries. In *ICDE*, pages 1566–1569, 2019.
[48] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, pages 214–225, 2004.
[49] Nyc tlc data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.
[50] *TNCs TODAY.* http://tncstoday.sfcta.org/.
[51] E. Tzirita Zacharatou, H. Doraiswamy, A. Ailamaki, C. T. Silva, and J. Freire. GPU rasterization for real-time spatial aggregation over arbitrary polygons. *PVLDB*, 11(3):352–365, 2017.
[52] E. Tzirita Zacharatou, A. Kipf, I. Sabek, V. Pandey, H. Doraiswamy, and V. Markl. The case for distance-bounded spatial approximations. In *CIDR*. http://cidrdb.org, 2021.
[53] *Uber Movement.* https://movement.uber.com/.
[54] F. van Diggelen and P. Enge. The world's first GPS MOOC and worldwide laboratory using smartphones. In *Proc. ION GNSS+*, pages 361–369, 2015.
[55] D. Vorona, A. Kipf, T. Neumann, and A. Kemper. DeepSPACE: Approximate geospatial query processing with deep learning. In *SIGSPATIAL/GIS*, pages 500–503. ACM, 2019.
[56] T. Zäschke, C. Zimmerli, and M. C. Norrie. The ph-tree: a space-efficient storage structure and multi-dimensional index. In *SIGMOD*, pages 397–408. ACM, 2014.

Indoor Spatial Queries: Modeling, Indexing, and Processing

Tiantian Liu[†] Huan Li[†] Hua Lu[‡] Muhammad Aamir Cheema[§] Lidan Shou[¶]
[†]Department of Computer Science, Aalborg University, Denmark
[‡]Department of People and Technology, Roskilde University, Denmark
[§]Faculty of Information Technology, Monash University, Australia
[¶]College of Computer Science, Zhejiang University, China
{liutt,lihuan}@cs.aau.dk, luhua@ruc.dk, aamir.cheema@monash.edu, should@zju.edu.cn

ABSTRACT

To support indoor spatial queries and indoor location-based services (LBS), multiple techniques including model/indexes and search algorithms have been proposed. In this work, we conduct an extensive experimental study on existing proposals for indoor spatial queries. We survey five model/indexes, compare their algorithmic characteristics, and analyze their space and time complexities. We also design an in-depth benchmark with real and synthetic datasets, evaluation tasks and performance metrics. Enabled by the benchmark, we obtain and report the performance results of all model/indexes under investigation. By analyzing the results, we summarize the pros and cons of all techniques and suggest the best choice for typical scenarios.

1 INTRODUCTION

Indoor location-based services (LBS) are becoming increasingly popular [6, 9]. Relevant applications, such as POI search [22, 28] and routing [11, 13, 14], are often built on top of typical spatial queries like range query, k nearest neighbor query, shortest path query, and shortest distance query. Therefore, the efficiency of processing such typical indoor spatial queries plays a key role in the success of indoor LBS.

To facilitate query processing for indoor LBS, space models, indexes and algorithms have been proposed. They all deal with indoor entities, e.g., rooms, doors, walls and floors. These entities form distinct topology that determines indoor distances and impacts indoor movement. As a result, the distances in indoor spatial queries must be measured appropriately, e.g., without involving straight line segments through walls. Also, indoor routing in shortest path/distance queries must consider connectivity and reachability between indoor locations.

To support indoor distance computation, existing models and indexes [27, 31, 37, 38] employ different approaches to integrate the geometry and topology information of an indoor space. Though all these approaches can be used to process the aforementioned indoor spatial queries, a comprehensive experimental study on all these proposals is still missing. Consequently, indoor LBS application developers inevitably encounter difficulties in choosing the appropriate technique for a given indoor space scenario.

To bridge this gap for LBS application development and disclose insights for further research on indoor data management, we conduct a comprehensive experimental study in this work. Our study focuses on five existing model/indexes that support typical indoor spatial queries on static indoor objects (e.g., POIs) or indoor shortest paths/distances. We compare the five proposals theoretically and empirically. Our contributions are as follows.

- We survey the five proposals by scrutinizing their structures, algorithmic characteristics, and space and time complexities.
- We design an in-depth benchmark with datasets, evaluation tasks, and performance metrics. The datasets consist of real and synthetic data characterized by distinctive indoor topology.
- Within the benchmark, we conduct extensive experiments to evaluate the performance of the five proposals in terms of construction cost and query efficiency.
- By analyzing the results, we disclose the pros and cons of the proposals, analyze the impact of different conditions, and recommend the best choice for typical application scenarios.

All code, data and test cases are open-sourced [1]. To the best of our knowledge, this work is the first that comparatively analyzes and evaluates the existing techniques under a unified framework.

The paper is organized as follows. Section 2 introduces indoor spatial queries and related work. Sections 3 and 4 present the indoor space model/indexes and query processing, respectively. Section 5 details the experimentation benchmark. Section 6 reports and analyzes the evaluation results. Section 7 concludes the paper.

2 INDOOR SPATIAL QUERIES

Table 1 lists the frequently used notations.

Table 1: Notations

Symbol	Meaning
$\mathbb{I}$	An indoor space
$p, q \in \mathbb{I}$	Indoor points
$o \in O$	A static indoor object
$d \in D$	A door
$v \in V$	An indoor partition
$\|p, q\|_I$	Indoor distance from p to q
$\langle p, d_i, \ldots, d_j, q \rangle$	An indoor path
$L(\phi)$	Length of a path ϕ

2.1 Indoor Space Concepts

Indoor space features distinct entities such as walls, doors, and rooms, which altogether form complex indoor topology that enables and constrains movements. Naturally, an indoor space is divided by walls and doors into **indoor partitions** like rooms, hallways or staircases. Two indoor partitions can be connected by a door or an open segment between them. Referring to the example floorplan in Figure 1, partitions 30 and 40 (denoted as v_{30} and v_{40}, respectively) are connected by an open segment d_3, In this paper, we refer to both doors and open segments as doors. We do not consider the width of a door and represent a door by its center point. In other words, each door can be generally regarded as an indoor point. Furthermore, a door can be unidirectional such as a security checkpoint at the airport. The door directionality makes the indoor distance between two points asymmetric. Referring to Figure 1, the shortest indoor path from p to p' and that from p' to p are different due to the unidirectionality of d_{12}.

Topology renders the indoor distance more complex than Euclidean distance. In Figure 1, the indoor distance $|p, o_1|_I$ from p to

o_1 is not subject to the straight line segment between them; it is the total length of the polyline $p \rightarrow d_{11} \rightarrow o_1$.

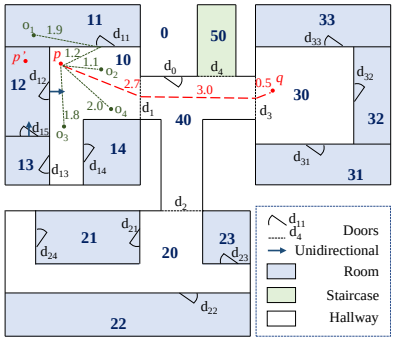


Figure 1: Example Floorplan

Lu et al. [27] proposes mappings to capture the relationships between indoor partitions and doors. In particular, $D2P_{\sqsupset}(d_i)$ gives the set of partitions that one can enter through door d_i and $D2P_{\sqsubset}(d_j)$ gives those that one can leave through door d_j. Besides, $D2P(d_i)$ gives a set of a partition pair (v_j, v_k) such that one can go through door d_i from partition v_j to v_k. Moreover, $P2D_{\sqsupset}(v_k)$ gives the set of enterable doors through which one can enter partition v_k, and $P2D_{\sqsubset}(v_k)$ gives the set of leaveable doors through which one can leave partition v_k. When doors are bidirectional, we use $P2D(v_k) = P2D_{\sqsupset}(v_k) \cup P2D_{\sqsubset}(v_k)$ to denote the set of doors associated to partition v_k.

Example 1. *In Figure 1, given the unidirectional door d_{12}, we have $D2P_{\sqsupset}(d_{12}) = \{v_{10}\}$, $D2P_{\sqsubset}(d_{12}) = \{v_{12}\}$, and $D2P(d_{12}) = \{(v_{12}, v_{10})\}$. Moreover, we have $P2D_{\sqsupset}(v_{12}) = \{d_{15}\}$, $P2D_{\sqsubset}(v_{12}) = \{d_{12}\}$, and $P2D(v_{12}) = \{d_{15}, d_{12}\}$.*

2.2 Indoor Spatial Query Types

We focus on *static* indoor objects such as POIs and facilities. Our study covers four fundamental indoor spatial query types.

Definition 1 (Range Query (RQ)). *Given an indoor point $p \in \mathbb{I}$, a set O of indoor objects, and a distance value r, a* range query *$\mathsf{RQ}(p, r)$ returns all indoor objects from O whose indoor distance from p is within r. Formally, $\mathsf{RQ}(p, r) = \{o \mid |p, o|_I \leq r, o \in O\}$.*

Definition 2 (k Nearest Neighbor Query (kNNQ)). *Given an indoor point $p \in \mathbb{I}$, a set O of indoor objects, and an integer value k, a k* nearest neighbor query *$k\mathsf{NNQ}(p)$ returns a set O' of k indoor objects whose indoor distances from p are the smallest, i.e., $|O'| = k$ and $\forall o_i \in O', o_j \in O \setminus O', |p, o_i|_I \leq |p, o_j|_I$.*

In Figure 1 where $O = \{o_1, \ldots, o_4\}$, a query $\mathsf{RQ}(p, 1.9\text{m})$ returns $\{o_2, o_3\}$ since the distances from p to o_1 and o_4 both exceed 1.9m.[1] Furthermore, a query $3\mathsf{NNQ}(p)$ returns $\{o_2, o_3, o_4\}$, since o_1's distance from p is the longest among all.

Definition 3 (Shortest Path Query (SPQ)). *Given a source point $p \in \mathbb{I}$, a target point $q \in \mathbb{I}$, a* shortest path query *$\mathsf{SPQ}(p, q)$ returns the shortest path $\phi = \langle p, d_i, \ldots, d_j, q \rangle$ from p to q such that 1) $d_i, \ldots, d_j$ are door sequences and each two consecutive doors are associated to the same partition, 2) p is in the partition having d_i as a leavable door, 3) q is in the partition having d_j as an enterable door, and 4) $\forall \phi'$ from p to q, $L(\phi) \leq L(\phi')$.*[2]

Definition 4 (Shortest Distance Query (SDQ)). *Given a source point $p \in \mathbb{I}$, a target point $q \in \mathbb{I}$, a* shortest distance query *$\mathsf{SDQ}(p, q)$ returns the shortest indoor distance from p to q, i.e., the length of $\mathsf{SPQ}(p, q)$.*

[1] Meter is the distance unit in all examples in this paper.

[2] $L(\phi) = \sum_{k=0}^{k=j} |d_k, d_{k+1}|_I$ where $d_0 = p$ and $d_{j+1} = q$.

As indicated by the red dashed polyline in Figure 1, a query $\mathsf{SPQ}(p, q)$ returns $\phi = \langle p, d_1, d_3, q \rangle$ as the shortest path from p to q, and the result of $\mathsf{SDQ}(p, q)$ is 2.7m + 3.0m + 0.5m = 6.2m.

2.3 Related Work

Indoor Space Modeling. Many indoor space models [7, 15, 20, 35, 36] focus on symbolic modeling of topological relationships between indoor partitions. Lacking of indoor distances, they cannot support the aforementioned distance-aware queries.

Indoor Moving Objects. Alamri et al. [4] propose an index tree for indoor moving objects based on connectivity between indoor cellular units. Kim et al. [19] propose to index indoor moving objects based on grid cells. Lin et al. [24] design an indoor moving object index to speed up complex semantic queries in multi-floor spaces. In the context of RFID indoor tracking, Yang et al. study continuous range monitoring queries [39] and probabilistic k nearest neighbor queries [40]. To improve the query result, Yu et al. [41] propose a particle filter-based method to infer the undetected locations of indoor moving objects. Assuming a probabilistic sample based location data format, Xie et al. [37, 38] process kNN query and range query for indoor moving objects. Considering uncertain object movements between observed time and query time, Li et al. [22] study searching the current top-k indoor dense regions. These works consider indoor moving objects with uncertain positions at a particular time. Unlike all these works on indoor moving objects, this study concerns spatial queries on static indoor objects, e.g., printers or ATMs.

Indoor Trajectories. Jensen et al. [15] study historical trajectories of RFID-tracked indoor objects. Delafontaine et al. [12] find sequential visiting patterns within historical Bluetooth tracking data. Given a past time or a time interval, Lu et al. define spatio-temporal joins [29] to find moving object pairs in the same indoor partition, and top-k queries [28] to find the most frequently visited indoor POIs. Ahmed et al. [2, 3] define threshold density query to find dense indoor semantic locations in a historical time interval. Assuming probabilistic sample based location records, Li et al. [23] find the top-k most popular indoor semantic regions with the highest object flow values. Jin et al. [17] study the similarity search over indoor trajectories, considering both spatial and semantic properties. By analyzing spatial constraints of indoor POIs, Jiang et al. [16] study the restoration of indoor trajectories. Li et al. [21] propose a coupled conditional Markov model to enrich indoor uncertain trajectories with mobility events and stay regions. Unlike these works, the queries studied in this paper focus on static objects or indoor paths.

Indoor Path Planning. Goetz and Zipf [14] study user-adaptive length-optimal indoor routing based on a weighted routing graph. Salgado et al. [30] study indoor keyword-aware skyline route query, considering the number of covered keywords and route distances. Feng et al. [13] study indoor keyword-aware routing queries to find shortest paths covering user-specified semantic keywords. Costa et al. [11] propose the context-aware indoor-outdoor path recommendation that minimizes the outdoor exposure and path distance. To enable navigation through movable obstacles, Sun et al. [33] study semantic assisted path planning over a gridded map of an indoor environment. Wang et al. [34] propose an obstacle-avoiding path planning algorithm to automate indoor robots. These techniques consider additional query semantics, and thus are different from the fundamental, pure shortest path/distance queries studied in this paper.

3 MODEL AND INDEXES

The aforementioned indoor spatial queries all involve indoor distances. To facilitate such queries, indoor distances must be considered in modeling and indexing indoor space.

3.1 Indoor Distance-Aware Model

Indoor distance-aware model [27] (IDMODEL) is a graph $G_{\text{dist}}(V, E_a, L, f_{\text{dv}}, f_{\text{d2d}})$. The first three elements capture indoor topology in an *accessibility base graph* $G_{\text{accs}}(V, E_a, L)$, where V is the set of vertexes each referring to an indoor partition, $E_a = \{(v_i, v_j, d_k) \mid d_k \in D, v_i \in D2P_{\sqsupset}(d_k) \wedge v_j \in D2P_{\sqsubset}(d_k)\}$ is a set of labeled, directed edges, and L is the set of edge labels each corresponding to a door in D. The additional two are mapping functions defined as follows.

$$f_{\text{dv}}(d_i, v_j) = \begin{cases} \max_{p \in v_j} ||d_i, p||_{v_j}, & \text{if } v_j \in D2P_{\sqsupset}(d_i); \\ \infty, & \text{otherwise.} \end{cases}$$

Here, $||p, q||_{v_j}$ is the indoor distance from a point p to a point q within the partition v_j. Note that $||p, q||_{v_j}$ is not necessarily a Euclidean distance because even within the same partition there may be obstacles in the line of sight between p and q. Specifically, *door-to-partition distance mapping* $f_{\text{dv}}(d_i, v_j)$ returns the longest distance one can reach within partition v_j from door d_i, if v_j is an enterable partition of d_i. Otherwise, it returns ∞.

$$f_{\text{d2d}}(v_j, d_i, d_j) = \begin{cases} ||d_i, d_j||_{v_j}, & \text{if } d_i \in P2D_{\sqsupset}(v_j) \\ & \text{and } d_j \in P2D_{\sqsubset}(v_j); \\ 0, & \text{if } d_i = d_j \\ & \text{and } d_i, d_j \in P2D(v_j); \\ \infty, & \text{otherwise.} \end{cases}$$

The *door-to-door distance mapping* $f_{\text{d2d}}(v_j, d_i, d_j)$ maps a partition v_j and two doors d_i and d_j to a distance value. If both doors are associated to v_j, it returns the distance from d_i to d_j within v_j, i.e., $||d_i, d_j||_{v_j}$. If d_i and d_j are identical and associated to v_j, we stipulate $f_{\text{d2d}}(v_j, d_i, d_j) = 0$. Otherwise, $f_{\text{d2d}}(v_j, d_i, d_j)$ returns ∞, indicating that one cannot go from d_i to d_j via v_j only.

Figure 2 illustrates the IDMODEL for the example shown in Figure 1. The outdoor space is captured in a special graph vertex v_0. Two hashmaps implement the mappings $f_{\text{dv}}(d_i, v_j)$ and $f_{\text{d2d}}(v_j, d_i, d_j)$. With directed edges, IDMODEL can support doors' directionality and temporal variation when needed.

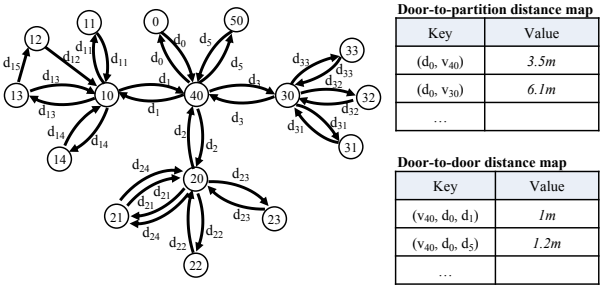


Figure 2: An Example of IDMODEL

With the two mappings $f_{\text{dv}}(d_i, v_j)$ and $f_{\text{d2d}}(v_k, d_i, d_j)$, a graph traversal algorithm [27] on IDMODEL is designed to compute the shortest door-to-door distance $\text{d2d}(d_s, d_t)$ from a source door d_s to a target door d_t. The basic idea is to keep expanding to unvisited doors based on the current shortest path until reaching the target door. Further, the shortest indoor distance from any point p to any point q can be computed by finding the minimum value of the distance summation $||p, d_p||_{v_p} + \text{d2d}(d_p, d_q) + ||d_q, q||_{v_q}$, where v_p and v_q are the partitions that host p and q, respectively, $d_p \in P2D_{\sqsubset}(v_p)$, and $d_q \in P2D_{\sqsupset}(v_q)$.

However, IDMODEL does not support fast determination of the host partition of a query/source point. It boils down to sequential scanning of all partitions if no additional index, e.g., R-tree, is used for the partitions. Also, to manage indoor static objects, IDMODEL needs additional object buckets each for a partition.

3.2 Indoor Distance-Aware Index

IDMODEL only captures the door-to-door and door-to-partition distances within a local partition, which entails extra search to compute the indoor distance for two points in different partitions.

To cut such costs, indoor distance-aware index [27] (IDINDEX) stores extra information on top of IDMODEL, namely, precomputed global door-to-door distances and their ordering in two matrices. The **door-to-door distance matrix** M_{d2d} is an N-by-N matrix where $\text{N} = |D|$ is the total number of doors and $M_{\text{d2d}}[d_i, d_j]$ gives the precomputed shortest indoor distance from d_i to d_j. The **distance index matrix** M_{idx} is also an N-by-N matrix such that $M_{\text{idx}}[d_i, k]$ gives the identifier of a door whose indoor distance from d_i is the k-th shortest among all the N doors.

The IDINDEX matrices for the top-left part in Figure 1 is illustrated in Figure 3. Here, we have $M_{\text{d2d}}[d_1, d_{15}] = 4.6\text{m}$. The first row of M_{d2d} shows that d_{15} has the longest indoor distance from d_1. Accordingly, we have $M_{\text{idx}}[d_1, 6] = d_{15}$ in M_{idx}.

$$\begin{pmatrix} & d_1 & d_{11} & d_{12} & d_{13} & d_{14} & d_{15} \\ d_1 & 0 & 1.7 & 2.7 & 3.6 & 2.8 & 4.6 \\ d_{11} & 1.7 & 0 & 1.9 & 3.6 & 2.8 & 4.6 \\ d_{12} & 2.7 & 1.9 & 0 & 2.6 & 1.8 & 1.6 \\ d_{13} & 3.2 & 3.4 & 2 & 0 & 2 & 1 \\ d_{14} & 2.8 & 2.8 & 1.8 & 1 & 0 & 2 \\ d_{15} & 4.3 & 3.5 & 1.6 & 1 & 2 & 0 \end{pmatrix} \qquad \begin{pmatrix} & 1 & 2 & 3 & 4 & 5 & 6 \\ d_1 & d_1 & d_{11} & d_{12} & d_{14} & d_{13} & d_{15} \\ d_{11} & d_{11} & d_1 & d_{12} & d_{14} & d_{13} & d_{15} \\ d_{12} & d_{12} & d_{15} & d_{14} & d_{11} & d_{13} & d_1 \\ d_{13} & d_{13} & d_{15} & d_{12} & d_{14} & d_1 & d_{11} \\ d_{14} & d_{14} & d_{13} & d_{12} & d_{15} & d_1 & d_{11} \\ d_{15} & d_{15} & d_{13} & d_{12} & d_{14} & d_{11} & d_1 \end{pmatrix}$$

(a) Distance Matrix M_{d2d} (b) Distance Index Matrix M_{idx}

Figure 3: An Example of IDINDEX

As the shortest indoor distances to all doors are precomputed and sorted for each door in IDINDEX, it is faster to compute the shortest indoor distance between any two points p and q in the indoor space. To support the shortest path query, in addition to the shortest distance value between any two points, IDINDEX also keeps the first-hop door of the corresponding shortest path. In this way, the complete shortest path between two points can be constructed by recursively concatenating the first-hop doors.

3.3 Composite Indoor Index

Composite indoor index [37] (CINDEX) is a layered structure for indexing indoor partitions and moving objects. It consists of three layers: geometric layer, topological layer, and object layer. In this study, we adapt the object layer to index static indoor objects. A partial example CINDEX for Figure 1 is given in Figure 4.

The **geometric layer** uses an R*-tree [8] to index all indoor partitions, with an additional skeleton tier to maintain the distances between staircases at different floors. To ease the geometrical computations, it decomposes each irregular partition[3] into regular ones using a decomposition algorithm [37]. Referring to the bottom-right of Figure 4, the hallway v_{10} is divided into two regular indoor partitions v_{10a} and v_{10b} by a door d_{16}. Afterwards, each regular partition is represented by a *Minimum Bounding Rectangle* (MBR). The MBRs are indexed by the R*-tree. As shown in the top-left of Figure 4, a non-leaf node R_1 is composed of six partitions in the leaf level, i.e., v_{10a}, v_{10b}, and v_{11}-v_{14}.

[3] A partition is irregular if it is non-convex or imbalanced (long in one dimension but short in the other).

The **topological layer** stores the connectivity information among indoor partitions, and it is integrated to the tree by inter-partition links. In particular, a leaf node v_i in the R*-tree is linked with a pointer record $(d_k, \uparrow v_j)$ to indicate that one can move from a partition v_i to another partition v_j through door d_k. As shown in the top-right of Figure 4, the two pointer records for v_{13} mean that v_{13} is adjacent to v_{10b} and v_{12} via d_{13} and d_{15}, respectively.

The **object layer** maintains a number of object buckets each for an indoor partition at the leaf node level of the R*-tree. Each indoor object o is kept in the bucket of the partition in which o is located. In addition, an object hashtable o-table : $O \rightarrow {}^*V$ maps each object to its host partition's pointer. Unlike [37, 38], the object buckets store static objects in this study. As shown in the bottom-left of Figure 4, the leaf node v_{10a} is linked to its object bucket with two static objects o_2 and o_4. Also, two corresponding records are kept in the object hashtable (o-table).

The R*-tree in CINDEX organizes partitions hierarchically, and thus enables search space pruning for distance relevant computations. As a result, CINDEX does not cache the precomputed door-to-door distances as IDINDEX does. Moreover, as the topological layer maintains the links between partitions and doors, which form an implicit graph structure, CINDEX does not need an explicit graph model to keep connectivity information. The topological layer's dynamic link updating makes CINDEX adaptive to possible temporal variations of doors.

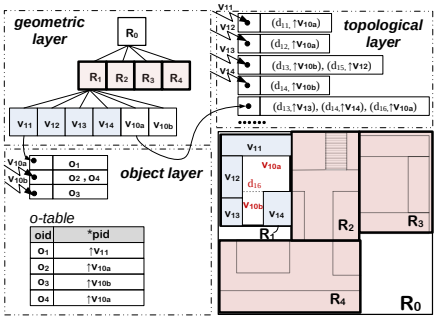


Figure 4: CINDEX Example (Adapted from [37])

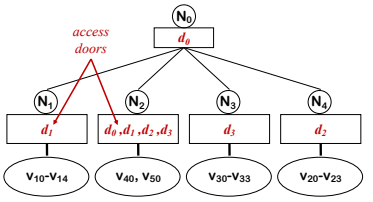
3.4 IP-Tree and VIP-Tree

Indoor partitioning tree [31] (IP-TREE) is a tree-based indoor partition index with a number of matrices each materializing the door-to-door distances within a local range. In particular, each leaf node of IP-TREE covers a number of topologically adjacent indoor partitions. The adjacent leaf nodes are combined to form a non-leaf node, and adjacent non-leaf nodes are combined hierarchically until a root node is formed. Each node N has a **distance matrix** and a number of **access doors**. An access door is a border door that connects N to its external space. AD(N) denotes N's access door set. The distance matrix for a leaf node stores the shortest distance (as well as the first-hop door on the shortest path) between every door of the leaf node to every access door of the leaf node. The distance matrix for a non-leaf node only stores the shortest distances and first-hop door between each pair of access doors of its child nodes. To compute the indoor distance from a point p to a point q, IP-TREE locates the lowest common ancestor of the leaf nodes `Leaf`(p) and `Leaf`(q), finds the access doors constituting the shortest path in that ancestor, and connects the materialized indoor distances involving p, the found access doors, and q.

Figure 5 shows an example of IP-TREE corresponding to Figure 1. The topologically adjacent partitions v_{10}-v_{14} form a leaf node N_1. Another leaf node N_2 is composed of partitions v_{40} and v_{50}. As N_1 and N_2 are connected by a border door d_1, d_1 is put into AD(N_1) and AD(N_2). For the leaf node N_1, the distance matrix stores the distances from each of its doors to the access door d_1 of N_1. For instance, the distance from N_1's only door d_{15} to access door d_1 contained by N_1 is 4.3m. Moreover, as the shortest path from d_{15} to d_1 is $\langle d_{15}, d_{12}, d_1 \rangle$, the first-hop door of the path is kept as d_{12} in the matrix. Differently, for the non-leaf node N_0, the distance matrix only keeps the distances between each pair of access doors. In the running example, each pair of access doors are directly connected. Therefore, no first-hop door is recorded. The storage space of each distance matrix will double when the door directionality needs to be considered, i.e., both the distances d2d(d_i, d_j) and d2d(d_j, d_i) are kept in each node.

As a variant of IP-TREE, vivid IP-Tree (VIP-TREE) [31] further accelerates the distance computation by materializing more precomputed information. Specifically, each leaf node N additionally maintains the shortest distance between each door contained by N and each access door in N's all ancestor nodes, along with the corresponding first-hop door information.

IP-TREE and VIP-TREE materialize a small number of distances only related to access doors that are critical in the overall topology of an indoor space. This design eases the on-the-fly distance related computations in spatial query processing.



	d_0	d_1	d_2	d_3
d_0	0	1.4	2	3.9
d_1	1.4	0	3	4
d_2	3.9	4	4.4	0
d_3	2	3	0	4.4

Distance Matrix for $\mathbf{N_0}$ (a non-leaf node)

	d_1	d_{11}	d_{12}	d_{13}	d_{14}	d_{15}
d_1	0	1.7	2.7	3.2	2.8	4.3, d_{12}

Distance Matrix for $\mathbf{N_1}$ (a leaf node)

Figure 5: An Example of IP-TREE

4 QUERY PROCESSING

All the aforementioned model/indexes can be used to process indoor spatial queries. Although query processing differs for different query types, all algorithms share a general paradigm as follows. First, an algorithm finds the initial indoor partition for a query. The initialization decides the indoor partition in which the query (or source) point p is located for a given RQ(p, r) (kNNQ(p), SPQ(p, q), or SDQ(p, q)). Subsequently, an algorithm expands from the initial partition, searching adjacent partitions via doors. Finally, the expansion stops when the search range is beyond the query range r for a RQ(p, r), or kNNs have been found for a kNNQ(p), or the target point q is met for a SPQ(p, q) or SDQ(p, q). Algorithms based on different model/indexes differ in their initializations and expansions. Below, we present a comprehensive analytical comparison of all model/indexes.

4.1 Algorithmic Comparison

Table 2 summarizes the comparison.

Distance Precomputation. IDMODEL and CINDEX do not precompute any indoor distances, whereas IDINDEX and IP-TREE/VIP-TREE maintain some door-to-door distances before query processing. In particular, IDINDEX precomputes the shortest indoor distances between every pair of doors, but IP-TREE/VIP-TREE only keeps a small number of distances in each tree node.

Model/Index Structure. IDMODEL is a labeled graph with distance mapping functions, whereas IDINDEX materializes two matrices for global door-to-door distances. Employing a tree-based structure, CINDEX keeps topological information incrementally by maintaining inter-partition links, whereas IP-TREE/VIP-TREE augments each tree node with a local distance matrix. More importantly, CINDEX forms the non-leaf tree nodes according to the

Table 2: Feature Comparison

Models	Precompute	Structure	Initialization	Expansion	RQ	kNNQ	SPQ	SDQ
IDModel	No	Graph+ Mappings	Sequential scan	Dijkstra	△	△	✓	✓
IDIndex	Yes	Matrix	Sequential scan	Loop	✓	✓	△	△
CIndex	No	Tree+Links	R*-Tree pruning	Dijkstra	✓	✓	△	△
IP-Tree	Yes	Tree+Matrix	Sequential scan	LCA	✓	✓	✓	✓
VIP-Tree	Yes	Tree+Matrix	Sequential scan	LCA	✓	✓	✓	✓

Table 3: Extensibility Analysis

	IDModel	IDIndex	CIndex	IP/VIP-Tree
Temporal Variation	✓	X	✓	X
Moving Objects	✓	✓	✓	✓
Uncertain Locations	X	X	✓	X
Keywords	✓	✓	✓	✓

Table 4: Complexity Analysis

	Space	RQ	kNNQ	SDQ	SPQ
IDModel	$O(\mathtt{V}+\mathtt{D}+2\mathtt{Vd}+\mathtt{Vd}^2)$	$O(\mathtt{oV}\log\mathtt{D})$	$O(\mathtt{oV}\log\mathtt{D})$	$O(\mathtt{V}\log\mathtt{D})$	$O(\mathtt{V}\log\mathtt{D}+\mathtt{w})$
IDIndex	$O(2\mathtt{D}^2)$	$O(\mathtt{od}\log\mathtt{D})$	$O(\mathtt{od}\log\mathtt{D})$	$O(\mathtt{d}^2)$	$O(\mathtt{d}^2+\mathtt{w})$
CIndex	$O(\mathtt{V}+\mathtt{Vd}+\mathtt{O})$	$O(\mathtt{oV}\log\mathtt{D})$	$O(\mathtt{oV}\log\mathtt{D})$	$O(\mathtt{V}\log\mathtt{D})$	$O(\mathtt{V}\log\mathtt{D}+\mathtt{w})$
IP-Tree	$O(\rho^2\mathtt{f}^2\mathtt{L}+\rho\mathtt{D})$	$O((\rho\log_\mathtt{f}\mathtt{L})^2(\mathtt{Vo}/\mathtt{L}+\rho))$	$O((\rho\log_\mathtt{f}\mathtt{L})^2(\mathtt{Vo}/\mathtt{L}+\rho))$	$O(\rho^2\log_\mathtt{f}\mathtt{L})$	$O((\rho^2+\mathtt{w})\log_\mathtt{f}\mathtt{L})$
VIP-Tree	$O(\rho^2\mathtt{f}^2\mathtt{L}+\rho\mathtt{D}\log_\mathtt{f}\mathtt{L})$	$O(\rho^2\log_\mathtt{f}\mathtt{L}(\mathtt{Vo}/\mathtt{L}+\rho))$	$O(\rho^2\log_\mathtt{f}\mathtt{L}(\mathtt{Vo}/\mathtt{L}+\rho))$	$O(\rho^2)$	$O(\rho^2+\mathtt{w})$

geometrical proximity of partitions, whereas IP-Tree/VIP-Tree do so based on the topological proximity of partitions.

Query Types. All model/indexes can support all the four query types. However, IDModel [27] does not provide RQ and kNNQ algorithms. Therefore, we implement the two algorithms and refer readers to the appendix in [26]. Also, there are no off-the-shelf SPQ/SDQ algorithms for IDIndex and CIndex. Nevertheless, the global door-to-door distances and the corresponding last-hop door information in IDModel can be used to expand path searching in SPQ/SDQ algorithms for IDIndex. For CIndex, the inter-partition links can be used for path expansion.

Initialization. To decide the initial indoor partition for a query, IDModel and IDIndex sequentially scan all partitions. Enabled by the R*-tree indexing partitions, CIndex can quickly find the host partition of any indoor point. In contrast, IP-Tree and VIP-Tree are based on pure topological relationships among partitions, and thus they also sequentially scan all partitions.

Expansion. As a graph-based model, IDModel expands to the next unvisited door in the spirit of Dijkstra's algorithm [18]. CIndex does so as well since the next-hop doors are captured in the inter-partition links on the topological layer. Instead of expanding via directly connected doors, IP-Tree/VIP-Tree finds the lowest common ancestor (LCA) node of p and q and locates the intermediate access doors on the shortest path straightforwardly. It is noteworthy that IDIndex alone cannot support topological door expansion. Instead, IDIndex relies on an underlying IDModel to loop through relevant indoor partitions' doors.

4.2 Complexity Analysis

Let V, D, O be the total number of indoor partitions, doors, and indoor objects, respectively. Let d and o be the average door number and average object number per partition, respectively. Let w be the average number of door nodes on a shortest path. For IP-Tree/VIP-Tree, we use f to denote the fan-out of the tree node, ρ the average access door number per node, and L the total number of leaf nodes. Table 2 summarizes the space complexity of all model/indexes and their time complexity for queries.

Space Complexity. IDModel $(V, E_a, L, f_{\mathrm{dv}}, f_{\mathrm{d2d}})$'s space complexity is $O(\mathtt{V}+\mathtt{Vd}+\mathtt{D}+\mathtt{Vd}+\mathtt{Vd}^2)=O(\mathtt{Vd}^2)$. IDIndex's space complexity is $O(2\mathtt{D}^2)=O(\mathtt{D}^2)$ as it consists of two door matrices. CIndex's space complexity is $O(\mathtt{V}+\mathtt{Vd}+\mathtt{O})=O(\mathtt{Vd}+\mathtt{O})$ where V, Vd, and O correspond to partition R*-tree, inter-partition links, and object hashtable, respectively. IP-Tree's space cost mainly consists of the distance matrices for leaf nodes and those for non-leaf nodes. The former's complexity is $O(\rho\mathtt{D})$ and the latter's is $O((\rho\mathtt{f})^2\mathtt{L})$ where ρf corresponds to the number of access doors from a child node and L reflects the number of non-leaf nodes. In contrast, VIP-Tree's space cost on the distance matrices for leaf nodes is $O(\rho\mathtt{D}\log_\mathtt{f}\mathtt{L})$, where $\log_\mathtt{f}\mathtt{L}$ corresponds to the ancestor number of each leaf node.

Time Complexity for RQ and kNNQ. RQ and kNNQ have similar time complexity as they both prune objects based on shortest distances. IDModel's search expands via qualified doors by graph traversal in $O(\mathtt{V}\log\mathtt{D})$ and iterates on the objects in each visited partition in $O(\mathtt{o})$. Also based on graph traversal, the search on CIndex obtains a subgraph in $O(\mathtt{V}\log\mathtt{D})$ and visits all objects in each partition of the subgraph in $O(\mathtt{o})$. IDIndex's search expands to the nearest partitions based on the sorted result in M_{idx}, and loops through each object in the expanded partition. So its time complexity is $O(\mathtt{od}\log\mathtt{D})$. The searches via IP-Tree and VIP-Tree work similarly. They prune a tree node based on its distance from the query point in $O(\log_\mathtt{f}\mathtt{L}\cdot\rho\cdot c)$, where c is the unit SDQ cost. Then, they qualify each object in the remaining nodes in $O(\log_\mathtt{f}\mathtt{L}\cdot\mathtt{V}/\mathtt{L}\cdot\mathtt{o}\cdot c)$. Given the SDQ complexity $O(\rho^2\log_\mathtt{f}\mathtt{L})$ for IP-Tree and $O(\rho^2)$ for VIP-Tree (to be detailed below), their RQ and kNNQ complexities are $O((\rho\log_\mathtt{f}\mathtt{L})^2(\mathtt{Vo}/\mathtt{L}+\rho))$ and $O(\rho^2\log_\mathtt{f}\mathtt{L}(\mathtt{Vo}/\mathtt{L}+\rho))$, respectively.

Time Complexity for SDQ and SPQ. For the graph traversal algorithms of IDModel and CIndex, the SDQ complexity is $O(\mathtt{V}\log\mathtt{D})$ and SPQ complexity is $O(\mathtt{V}\log\mathtt{D}+\mathtt{w})$ with additional cost to backtrack the shortest path in w hops. For IDIndex, the only cost of SDQ is to loop through two door sets corresponding to p and q by a complexity of $O(\mathtt{d}^2)$. The extra cost of SPQ to concatenate shortest path is of $O(\mathtt{w})$. For IP-Tree, SDQ needs to search the lowest common ancestor and then find a pair of access doors from that ancestor node, resulting in a complexity of $O(\rho^2\log_\mathtt{f}\mathtt{L})$. In contrast, VIP-Tree materializes the distances from a leaf node to each access door in the ancestors. Its SDQ complexity is $O(\rho^2)$. The additional cost to construct shortest path in SPQ is $O(\mathtt{w}\log_\mathtt{f}\mathtt{L})$ for IP-Tree and $O(\mathtt{w})$ for VIP-Tree.

4.3 Extensibility Analysis

Table 3 summarizes the extensibility of all model/indexes.

Temporal Variation. Indoor topology may feature temporal variations, e.g., doors have open and close hours. To support indoor spatial queries in such cases, temporal variations like open and close time of doors can be maintained as a table attached to the accessibility base graph of IDModel or the topological layer of CIndex [25]. However, frequent temporal variations are hard to handle for IDIndex and IP-Tree/VIP-Tree as they need to precompute door-to-door distances globally or locally.

Moving Objects. CIndex [37, 38] is designed for managing indoor moving objects. It supports distance-aware queries like kNNQ and RQ, and also distance-aware joins like semi-range join and semi-neighborhood join. All other model/indexes can also index moving objects by maintaining dynamic object buckets

attached to indoor partitions in a way similar to how we handle the static objects. Nevertheless, the buckets need to be updated appropriately for indoor moving objects.

Uncertain Locations. In some settings, indoor points or objects are represented as uncertain regions. To process indoor spatial queries over uncertain locations, a model/index should support geometric operations on partitions. As a result, only CINDEX with partition R*-tree excels at handling uncertain locations [37, 38].

Keywords. A spatial keyword query [10] returns objects or paths that are spatially and textually relevant to the user-specified location(s) and keyword(s). Such queries can be supported if we extend the model/indexes by additionally maintaining mappings between partitions/objects and keywords. Especially, top-k keyword-aware shortest path queries have been supported based on IDMODEL [13], and boolean kNN spatial keyword queries have been supported based on VIP-TREE [32].

5 BENCHMARK

In this section, we detail the benchmark for evaluating the indoor spatial query techniques (model/indexes and algorithms). All code, data, and test cases are available online [1].

5.1 Datasets

We use four very different indoor space datasets, each featuring a distinctive indoor topology. The floorplans are briefly represented and illustrated in Figure 6. The data statistics are given in Table 5.

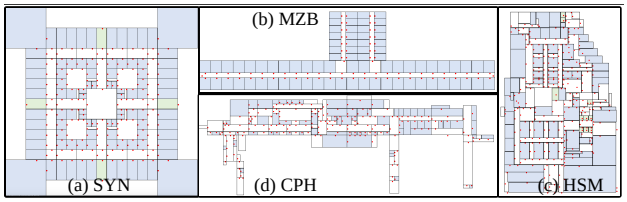


Figure 6: Floorplan of Datasets.

Synthetic Building (**SYN**) is a n-floor building. Its each floor is from a real-world floorplan [4] of 1368m × 1368m with 141 partitions and 216 doors. Its each two adjacent floors are connected by four 20m long stairways. By default, we set $n = 5$ and get the default dataset SYN5. To study the effect of topological changes, from SYN5 we derived SYN5$^-$ with fewer doors and SYN5$^+$ with more doors. Note that varying the door number will significantly change the connectivity and accessibility of the partitions, leading to a major topological change. We also form SYN5^0 in which the hallways are not decomposed [5].

Menzies Building (**MZB**) [6] is a landmark building at Clayton campus of Monash University. Each floor takes approximately 125m × 35m and connects to adjacent floors by two or four stairways each being 5m long. In total, there are 1344 partitions (including 34 staircases and 85 hallways) and 1375 doors. By changing the hallway decomposition, we form MZB^0 in which the hallways are not decomposed and MZB$^\Delta$ in which the hallways are decomposed into more partitions than default.

Hangzhou Shopping Mall (**HSM**) is a 7-floor mall in Hangzhou, China, occupying 2700m × 2000m. Ten stairways connect each two adjacent floors. Each floor contains 150 partitions and 299 doors on average. In total, there are 1050 partitions (including 70 staircases and 133 hallways) and 2093 doors.

[4] https://deviantart.com/mjponso/art/Floor-Plan-for-a-Shopping-Mall-86396406

[5] We precompute the door-to-door distance matrix for each hallway when it is not decomposed. The hallways are of irregular and concave shapes, and thus the door-to-door distance in a hallway can not use the Euclidean distance.

[6] https://www.monash.edu/virtual-tours/menzies-building

Copenhagen Airport (**CPH**) refers to the ground floor of Copenhagen Airport [7], taking around 2000m × 600m with 147 partitions (including 25 hallways) and 211 doors.

Overall Analysis of Different Datasets. The statistics of the datasets are given in Table 5. We use $\#dv$ to denote the number of doors in a partition, and conduct quartile statistics [5] on $\#dv$. In Table 5, Q1($\#dv$), Q2($\#dv$), and Q3($\#dv$) denote the first, second, third quartiles of $\#dv$, respectively, and max($\#dv$) denotes the maximum value of $\#dv$. In addition, we also plot the distributions of $\#dv$ over all partitions in each dataset in Figure 7.

Table 5: Statistics of Datasets

Datasets	SYN	MZB	HZM	CPH	SYN5$^-$	SYN5$^+$	SYN5^0	MZB^0	MZB$^\Delta$
Floors	n	17	7	1	5	5	5	17	17
Doors	216n	1375	2093	211	840	1280	880	1308	1480
Partitions	141n	1344	1050	147	705	705	505	1276	1449
Hallways	41n	85	483	72	205	205	5	17	190
C-Pars	8n	52	133	20	20	40	5	19	157
Length(m)	1368	125	2700	2000	1368	1368	1368	125	125
Width(m)	1368	35	2000	600	1368	1368	1368	35	35
Q1($\#dv$)	2	1	2	1	1	2	1	1	1
Q2($\#dv$)	2	1	4	2	1	3	2	1	1
Q3($\#dv$)	4	1	5	4	3	4	3	1	1
max($\#dv$)	10	56	17	12	10	10	132	82	47

Based on the space scale information and door distribution information from Table 5 and Figure 7, we summarize the characteristics of each dataset as follows.

- SYN: The overall space is square and regular. The number of doors and partitions in each floor is medium (216 doors and 141 partitions per floor). The door density within each partition is small (with Q2 equals only 2).
- MZB: The overall space is long and narrow with large scale crucial partitions (*C-Pars* for short). The number of doors and partitions in each floor is relatively small (80.4 doors and 76.8 partitions on average), whereas the overall size of doors and partitions is large due to the floor number. The planning of doors is rather skewed in that most partitions have only 1 or 2 doors while there are some C-Pars that accommodate 56 doors (as shown in Figure 7(b)).
- HSM: The overall space is long and relatively narrow. The number of doors and partitions in each floor is medium and the overall size of doors and partitions is large. The planning of doors is regular and door density in each partition is medium (Q2 and Q3 are equal to 4 and 5, respectively).
- CPH: The space is long, narrow yet open, resulting in a small number of doors and partitions. The door distribution is regular and door density in each partition is small (Q2 equals 2).

5.2 Object/Query Workload Generation

For each dataset, we randomly generated a set O of valid points as static objects, each object in O falling in an indoor partition. To test the effect of different object numbers, we vary $|O|$ as 500, 1000, 1500, 2000 and 2500.

The augment generation for each query type is detailed below.

RQ(p, r). We vary the range value r according to the predefined values in Table 6 (default values in bold). For each r, we generate ten RQ instances with a random p in the indoor space.

kNNQ(p). Similar to RQ generation, we generate ten random kNNQ instances for each k value given in Table 6.

As SPQ and SDQ can be integrated into one search procedure, we use SPDQ(p, q) to denote the integrated query that returns the shortest path from p to q along with the corresponding shortest distance value. In the following sections, we evaluate search performance of SPDQ only.

[7] https://www.cph.dk/en/practical

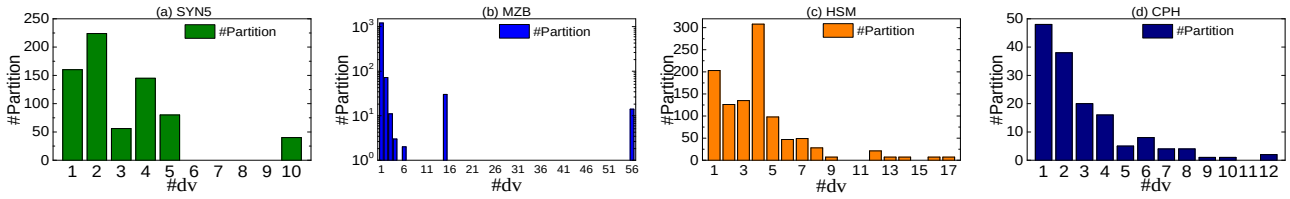


Figure 7: Distribution of $\#dv$ (number of doors in a partition) on (a) SYN5, (b) MZB, (c) HSM, and (d) CPH.

Table 6: Evaluation Settings (Default Parameters in Bold)

Symbol & Meaning		Task	Metrics	Queries	Dataset	Parameter Setting
n	floor number	A B1	a1, a2 b1, b2, b3 (only for SPDQ)	- RQ, kNNQ, SPDQ	SYN	3, **5**, 7, 9
$\|O\|$	object number	B2	b1, b2	RQ, kNNQ	all	500, 1000, **1500**, 2000, 2500
r	range value	B3	b1, b2	RQ	SYN5, HZM, CPH MZB	200, 400, **600**, 800, 1000 20, 40, **60**, 80, 100
k	-	B4	b1, b2	kNNQ	all	1, 5, **10**, 50, 100
s2t	source-target distance	B5	b1, b2, b3	SPDQ	SYN5, HZM, CPH MZB	1100, 1300, **1500**, 1700, 1900 30, 60, **90**, 120, 150
-	topological change	B6	b1, b2, b3 (only for SPDQ)	RQ, kNNQ, SPDQ	SYN	SYN5$^-$, SYN5, SYN5$^+$
-	decomposition method	B7	b1, b2, b3 (only for SPDQ)	RQ, kNNQ, SPDQ	SYN MZB	SYN5^0, SYN5 MZB^0, MZB, MZB$^\Delta$

SPDQ(p, q). We use a parameter s2t to control the shortest distance from the source p and target q. Its parameter values are listed in Table 6. For each s2t, we generate ten different (p, q) pairs to form SPDQ instances as follows. First, we randomly select an indoor point p and find a door d whose indoor distance from p approximates s2t. Next, we expand from d to find a random point q whose indoor distance from p approximates s2t.

5.3 Model/Index Settings

IDMODEL. For each partition v_i, we implemented the door-to-door distance mapping $f_{\text{d2d}}(v_i, \cdot, \cdot)$ as a 2D array, and door-to-partition distance mapping $f_{\text{dv}}(\cdot, v_i)$ as an 1D array. Besides, the partition mappings $P2D_{\sqsupset}(v_i)$ and $P2D_{\sqsubset}(v_i)$ (cf. Section 2.1) were implemented as lists associated to v_i. Moreover, the door mappings $D2P(d_i)$, $D2P_{\sqsupset}(d_i)$, and $D2P_{\sqsubset}(d_i)$ were implemented as lists associated to the door d_i.

IDINDEX. The distance matrix and distance index matrix were implemented as 2D arrays.

CINDEX. Since the partitions in the datasets rarely intersect, we used an R-tree instead of R*-tree to index partitions while preserving roughly the same spatial search performance. We set the tree fan-out to 20 as suggested in a previous work [37]. Each partition's inter-partition links were maintained in an inner list.

IP-TREE and VIP-TREE. We set the minimum fanout to 2 for non-leaf tree nodes, as suggested in [31]. As each leaf node maintains the shortest distance for each pair of doors in it, the computation will be complicated if a leaf node contains too many C-Pars that each has many doors. Following work [31], we designate that each leaf node can only contain one crucial partition and regard a partition as crucial partition if its door number exceeds a threshold γ. Through tuning, we got optimal γ as 6, 4, 7, and 5 for SYN, MZB, HZM, and CPH, respectively.

5.4 Performance Evaluation Procedure

Concerning model construction and query processing, the following tasks are implemented to evaluate each model/index. For each task, a parameter is varied with others fixed to default. Table 6 lists all the evaluation settings. The code of following evaluation procedures and their query instances are also available online [1].

A **Model Construction.** For each model/index, we evaluate its (**a1**) model/index size and (**a2**) construction time. In this task, we vary the number of floors in synthetic datasets.

B **Query Processing.** We evaluate the search efficiency of a given query type. The metrics are (**b1**) running time, (**b2**) memory use, and (**b3**) number of visited doors (NVD) for SPDQ.

B1 **Effect of Floor Number n.** Using SYN with floor number n varied from 3 to 9, we test the search efficiency for each indoor spatial query algorithm.

B2 **Effect of Object Number $|O|$.** To test RQ and kNNQ, we vary $|O|$ from 500 to 2500 in all datasets.

B3 **Effect of Range Distance r.** We vary and test the augment r of RQ. In particular, we vary r from 200m to 1000m in SYN5, HZM and CPH, and from 20m to 100m in MZB.

B4 **Effect of k.** We vary and test kNNQ's augment k from 1 to 100 in all datasets.

B5 **Effect of Source-Target Distance s2t.** To test SPDQ, we vary s2t from 1100m to 1900m in SYN5, HZM, and CPH, and from 30m to 150m in MZB.

B6 **Effect of Topological Change.** We vary indoor topology by changing the door number from 840 to 1280 in SYN5 and obtain SYN5$^-$ and SYN5$^+$.

B7 **Effect of Hallway's Decomposition Method.** We use SYN5 and MZB with the derived datasets, SYN5^0, MZB^0 and MZB$^\Delta$.

6 RESULTS ANALYSIS

This section reports and analyzes the experimental results. All experiments are implemented in Java and run on a MAC with a 2.30GHz Intel i5 CPU and 16 GB memory.

6.1 Model/Index Construction

We vary the floor number n on SYN and obtain four variants SYN3, SYN5, SYN7, and SYN9. We construct the five model/indexes (cf. Section 3) and report their size and construction time in Figures 8 and 9. The cost of maintaining static objects is excluded as it is the same for all model/indexes.

- According to the results on SYN3 to SYN9 in Figure 8, each model/index's size increases steadily with a larger floor number.

When there are more doors and partitions, more storage space is needed to handle the indoor space.

- Among all, IDMODEL construction requires the least costs on storage (Figure 8) and time (Figure 9). This is because IDMODEL is extended based on a simple graph model and maintains only a small amount of geometric information locally. For large-scale and complex-topology spaces (e.g., SYN9, MZB, and HZM), IDMODEL has clearer advantages over the tree-based indexes (i.e., IP-TREE and VIP-TREE).
- As expected, IDINDEX always takes much time and storage to construct due to its global door-to-door distance precomputation. When there are many doors, it is difficult to fit the corresponding matrices in memory. In comparison, IP-TREE and VIP-TREE precompute less information and therefore their consumptions on time and storage are medium.
- In addition to maintaining the topology, CINDEX needs to construct a partition R-tree. Therefore, it incurs extra time and space overheads compared to IDMODEL.

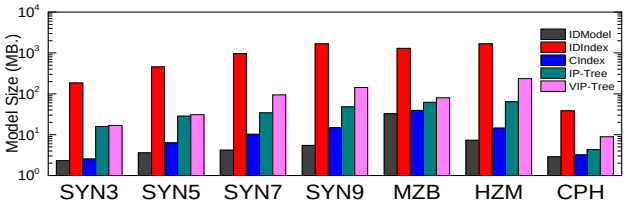


Figure 8: Model Size

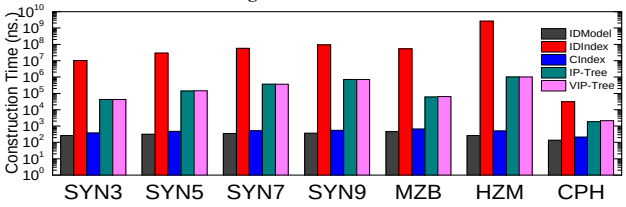


Figure 9: Construction Time

6.2 Query Processing

All results are averaged over 10 queries (cf. Section 5.2).

B1 Effect of Floor Number n (using SYN)

RQ and kNNQ: The query time and memory use for RQ are reported in Figures 10 and 11, respectively, and those for kNNQ are reported in Figures 12 and 13, respectively.

- For both query types, IDINDEX always runs fastest as shown in Figures 10 and 12, unaffected by the varying floor number n. The price behind this is to maintain the memory-resident distance matrices, which increases rapidly with n. Referring to Figures 11 and 13, when n grows to 9, IDINDEX requires up to 1600MB of memory on both queries.
- On each SYN dataset, IP-TREE and VIP-TREE need more time to complete the two queries. Through analysis, we found that the two indexes need to prune tree nodes when searching for qualified objects. In the absence of global door-to-door distances, they need a lot of on-the-fly calculations to get the shortest distance from a query point to a tree node. Being consistent with the complexity analysis in Table 4, VIP-TREE outperforms IP-TREE for both queries. However, due to the good scalability of the tree structure, both indexes' running time is relatively stable as shown in Figures 10 and 12.
- IDMODEL and CINDEX perform similarly, and their execution time increases with a larger n (Figures 10 and 12). When n increases, IDMODEL has a slight advantage as CINDEX costs more time in space pruning. In terms of memory overhead, the two indexes are almost the same.

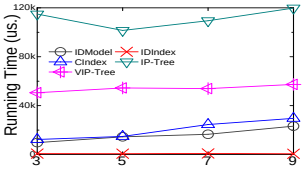


Figure 10: RQ Time vs. n

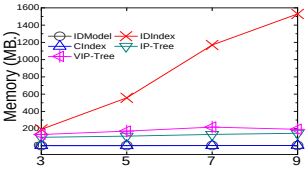


Figure 11: RQ Memory vs. n

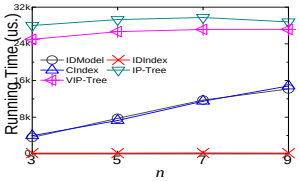


Figure 12: kNNQ Time vs. n

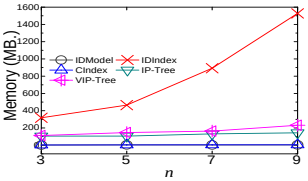


Figure 13: kNNQ Memory vs. n

SPDQ: The running time, memory use, and number of visited doors (NVD) are reported in Figures 14, 15, and 16, respectively.

- IDINDEX's running time and NVD are insensitive to the increasing floor number n. However, its memory use grows moderately as n increases. In the case of SPQ and SDQ, we recommend using IDINDEX when the door size is relatively small.
- In contrast to IDINDEX, the memory of IDMODEL and CINDEX is relatively stable (Figure 15), and their query performance deteriorates as the space scale increases (Figure 14).
- IP-TREE and VIP-TREE achieve clearly good performance on SPDQ, in both running time and memory use. Unlike IDINDEX that precomputes global door-to-door distances or IDMODEL and CINDEX that compute distances on the fly, IP-TREE and VIP-TREE cache relevant distance information only for those access doors on shortest paths. Thus, without degrading query performance, they only incur slightly more memory overhead than IDMODEL and CINDEX (Figures 14 and 15).

B2 Effect of Object Number $|O|$

RQ: With different sizes of O, the running time and memory use are reported in Figures 17 and 18, respectively.

- Algorithms based on different model/indexes are almost insensitive to $|O|$ in running time, implying that each can prune irrelevant objects effectively and stop searching early. A larger $|O|$ results in higher object density. This tends to increase the query processing time in general, as the query algorithms need to process larger object buckets. However, this impact is negligible according to the results in Figure 17. This implies that all model/indexes are good at pruning indoor partitions and thus object buckets when processing RQ.
- Referring to Figure 17, IDINDEX runs faster than others by several orders of magnitude in all datasets, thanks to its precomputed global door-to-door distances. However, it also requires memory an order of magnitude higher to store the distance matrix (Figure 18). A special case occurs on CPH (Figure 18(d)) that IP-TREE and VIP-TREE consume more memory than others. First, the door number of CPH is quite small such that the matrices of IDINDEX are not large. Second, as there are fewer access doors, IP-TREE/VIP-TREE involves heavy on-the-fly computations on distances between doors and non-leaf nodes and thus needs more memory for the intermediate results.
- On each dataset, IDMODEL and CINDEX incur almost the same execution time (see Figure 17), as they both use graph traversal to search for objects. Under complex indoor topology, CINDEX using R-tree does not have much advantage in spatial pruning.
- IP-TREE and VIP-TREE perform differently on different datasets. They outperform IDMODEL and CINDEX on MZB but are

worse on the others (see Figure 17). Recall that MZB features some C-Pars having up to 56 doors. In such a case, the efficiency of graph traversal is much lower than searching on the tree structure. On the contrary, when the number of candidate doors for the next hop is relatively small, the graph-based search algorithms are advantaged in range queries. Therefore, we recommend using IP-TREE/VIP-TREE to perform RQ in spaces with very large main corridors.

- Referring to Figure 17, VIP-TREE is generally faster than IP-TREE because of more cached distances. IP-TREE needs to compute more intermediate results on the fly. However, memory use is close between the two (see Figure 18).

*k*NNQ: Figures 19 and 20 report $|O|$'s impact on the time and memory costs, respectively. In general, each model/index's performance on *k*NNQ exhibits similar trend as that on RQ.

- Referring to Figure 19, the time cost of each algorithm on each dataset remains stable, showing that large object workloads (and high object density) have little effect on all models.
- On datasets with relatively large numbers of doors and partitions (i.e., SYN5, MZB, and HSM), IDINDEX runs faster by orders of magnitude. However, its memory use is clearly larger.
- On one-floor CPH with small numbers of doors and partitions, IP-TREE and VIP-TREE incur more running time as well as higher memory use (Figures 19(d) and 20(d)). However, they run faster on MZB (Figure 19(b)) in which many access doors exist due to many C-Pars (see Table 5).
- IDMODEL and CINDEX perform comparably as shown in Figures 19 and 20. Without a specially designed partition R-tree, IDMODEL achieves quite good object pruning due to the efficient distance mapping maintained in its edges and vertexes.

B3 Effect of Range Distance *r*

RQ: The time and memory costs with respect to varied *r* are reported in Figures 21 and 22, respectively.

- On SYN5, MZB, and HSM with complex indoor topology, IDINDEX's running time reported in Figure 21 increases slowly with a growing *r*. In contrast, on the simple-topology CPH, the advantage of IDINDEX over others is not marked.
- IDMODEL and CINDEX perform well on all datasets, except on MZB (Figure 21(b)) that has a large number of C-Pars. This again reflects the disadvantages of the graph-based traversal algorithms when dealing with this particular topology type. Nevertheless, through efficient node search and on-the-fly distance computation, these two model/indexes always have the smallest memory overhead.
- When increasing *r*, the running time of IP-TREE and VIP-TREE in Figure 21 increase steadily on all datasets. A larger *r* needs to consider a tree node farther from the node where the query point is located, and thus introduces more computations on the distance from a door to some non-leaf nodes. As the distance to the access door of each ancestor node is materialized at the leaf node, VIP-TREE runs faster than IP-TREE.

B4 Effect of *k*

*k*NNQ: The time and memory costs with respect to different *k* values are reported in Figures 23 and 24, respectively.

- Similar to increasing *r* value in RQ, increasing *k* leads to more search time by each model/index according to the results reported in Figure 23. Among them, IDINDEX's running time increases slowest. In addition, IP-TREE/VIP-TREE show exponential growth on SYN, HSM, and CPH. This is because the two indexes need to access the topologically far-away partitions and compute the distances to them on the fly when *k* is large.
- Considering both time and memory costs, IDMODEL and CINDEX achieve a good balance when searching for nearest neighbor objects (see Figures 23 and 24).

B5 Effect of Source-Target Distance s2t

SPDQ: The time cost, memory use, and NVD for different s2t values are reported in Figures 25, 26, and 27, respectively.

- IDINDEX runs the fastest and is not affected by s2t as reported in Figure 25. As only a small number of doors are required to process after the source point and before the target point, its NVD is always small (Figure 27). Nevertheless, its global distance matrix takes up a lot of memory (Figure 26).
- IDMODEL and CINDEX use the same graph search process. Note that because the Euclidean distance is no larger than the indoor distance, using R-tree to prune space by Euclidean distance does not really reduce the number of doors to visit. Therefore, the two models' NVDs in Figure 27 are almost the same. Also, as s2t increases, the candidate space becomes larger and the running time of the two becomes longer (see Figure 25).
- On MZB and HSM (Figure 25(b) and (c)), VIP-TREE achieves query performance comparable to IDINDEX that precomputes door-to-door distances. Both MZB and HSM are large-scale and have over 1000 doors. In the routing process based on VIP-TREE, the precomputed distances in non-leaf nodes greatly accelerate the expansion to the target point. Therefore, VIP-TREE is particularly suitable for the shortest path search in indoor spaces with complex structures.

Table 7: Results of RQ with Topological Change

Model	Time (us.)			Memory (MB.)		
	SYN5$^-$	SYN5	SYN5$^+$	SYN5$^-$	SYN5	SYN5$^+$
IDMODEL	11111	12770	19893	2	4	4
IDINDEX	308	417	910	289	520	809
CINDEX	13697	14877	19285	4	3	4
IP-TREE	29004	136600	574069	85	95	194
VIP-TREE	18008	58369	195583	75	171	220

Table 8: Results of *k*NNQ with Topological Change

Model	Time (us.)			Memory (MB.)		
	SYN5$^-$	SYN5	SYN5$^+$	SYN5$^-$	SYN5	SYN5$^+$
IDMODEL	5939	8180	10051	2	3	3
IDINDEX	165	146	181	469	573	1053
CINDEX	6865	8476	12998	4	4	4
IP-TREE	17341	36626	107798	74	89	139
VIP-TREE	14535	30145	75439	78	145	146

Table 9: Results of SPDQ with Topological Change

Model	Time (us.)			Memory (MB.)			NVD		
	SYN5$^-$	SYN5	SYN5$^+$	SYN5$^-$	SYN5	SYN5$^+$	SYN5$^-$	SYN5	SYN5$^+$
IDMODEL	23009	33213	35522	58	59	91	6946	10074	11426
IDINDEX	40	65	79	182	416	748	6	8	9
CINDEX	20219	31635	40408	51	63	99	6946	10074	11426
IP-TREE	3717	6398	7252	43	44	74	236	843	1455
VIP-TREE	2349	2369	2493	55	43	105	52	61	90

B6 Effect of Topological Change

RQ and *k*NNQ: The time cost and memory use with respect to topology characteristics are reported in Tables 7 and 8 respectively.

- IDINDEX runs fastest, but it needs large memory to store the door-to-door distance matrix. With increasing number of doors, its time cost and memory use increase steadily.
- IDMODEL and CINDEX use the smallest memory when processing RQ and *k*NNQ. Regarding the time cost, they perform medium. When the topology becomes more complex, the memory use keeps stable and the time cost increases slightly.
- IP-TREE and VIP-TREE cost more time to process RQ and *k*NNQ. Moreover, when the topology becomes more complex, the time cost rises rapidly. E.g., RQ's time cost using IP-TREE grows nearly 20 times from SYN5$^-$ to SYN5$^+$.

SPDQ: The time cost, memory use and NVD with respect to different topology characteristics are reported in Table 9.

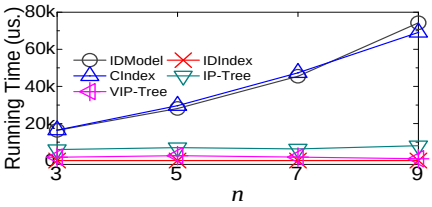


Figure 14: SPDQ Time vs. n

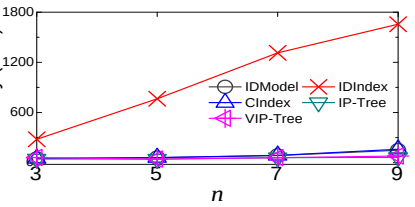


Figure 15: SPDQ Memory vs. n

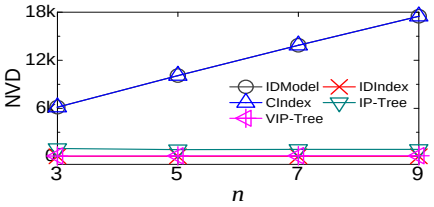


Figure 16: NVD in SPDQ vs. n

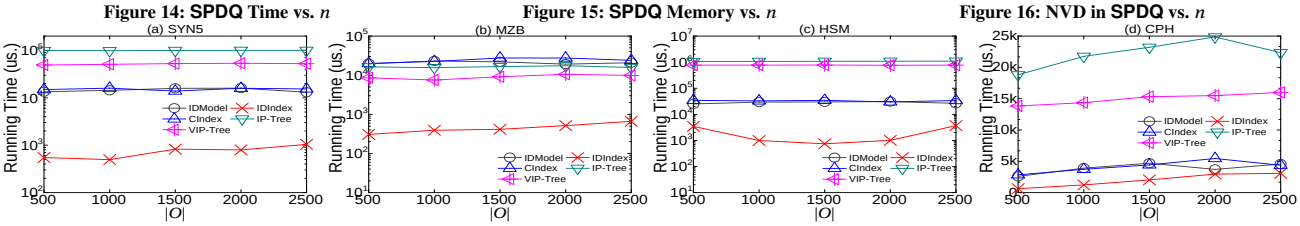


Figure 17: RQ Time vs. $|O|$

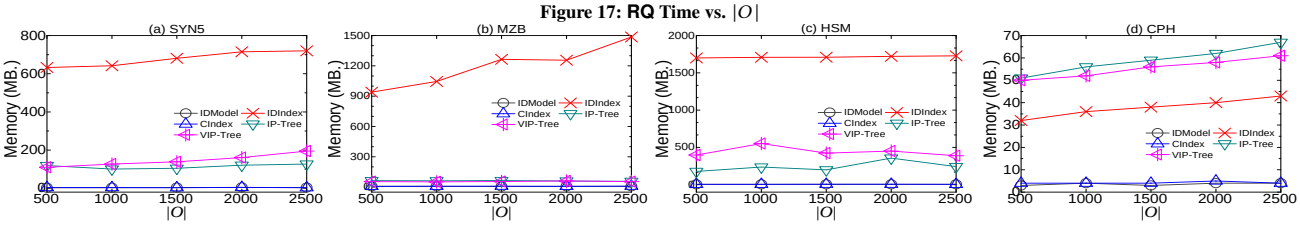


Figure 18: RQ Memory vs. $|O|$

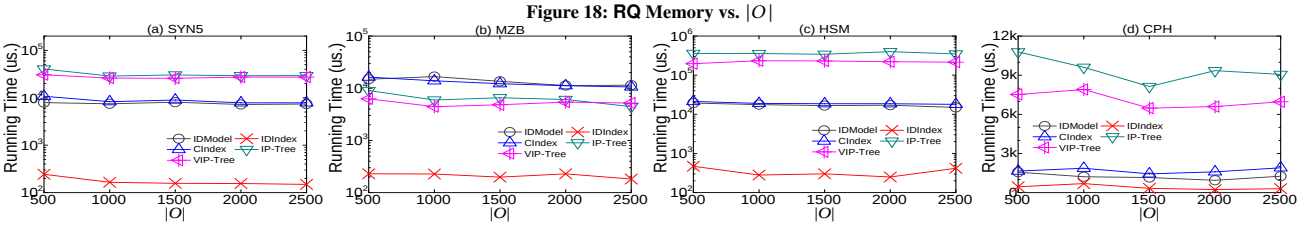


Figure 19: kNNQ Time vs. $|O|$

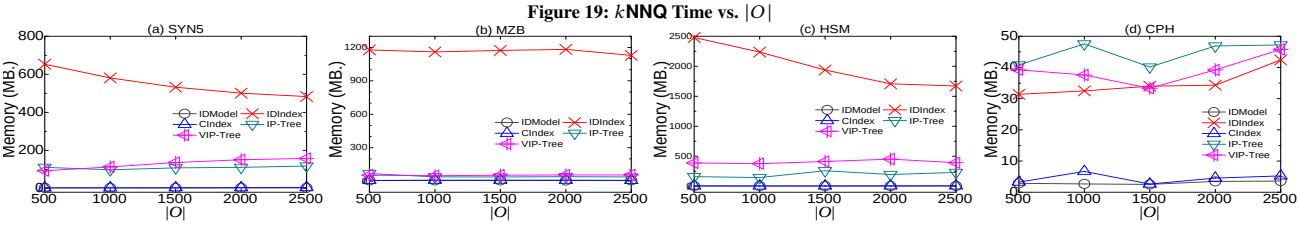


Figure 20: kNNQ Memory vs. $|O|$

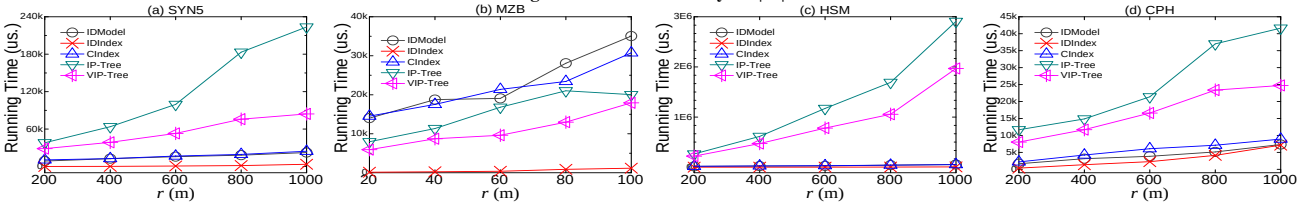


Figure 21: RQ Time vs. r

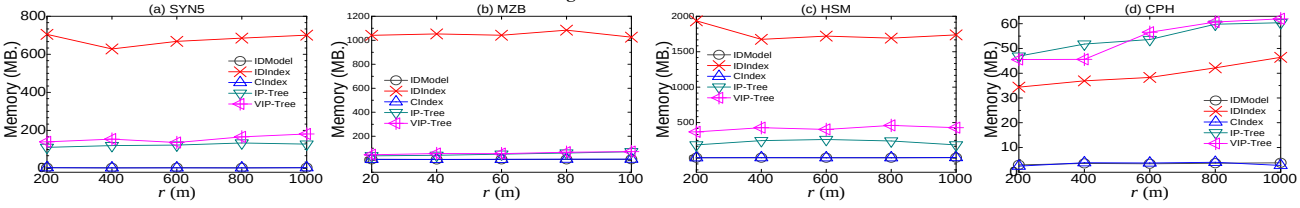


Figure 22: RQ Memory vs. r

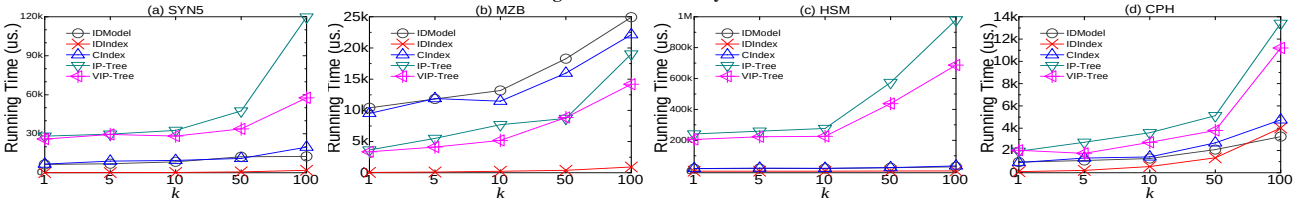


Figure 23: kNNQ Time vs. k

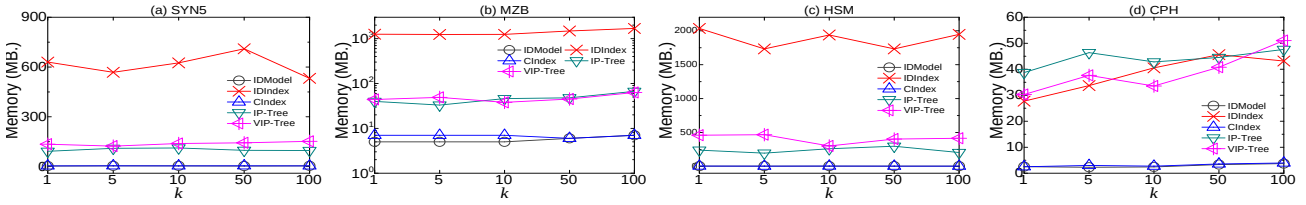


Figure 24: kNNQ Memory vs. k

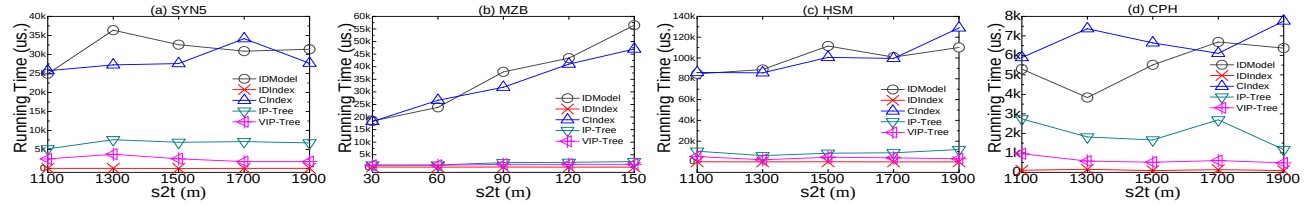


Figure 25: SPDQ Time vs. s2t

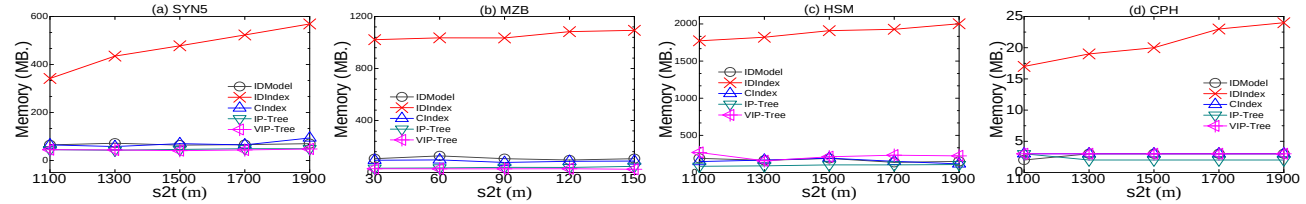


Figure 26: SPDQ Memory vs. s2t

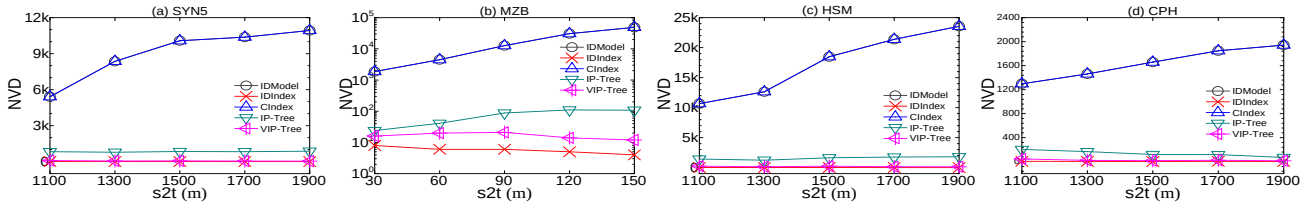


Figure 27: NVD in SPDQ vs. s2t

- Like in the other cases, IDINDEX performs best in terms of the time cost but costs most memory compared with others. When the topology becomes complex, IDINDEX's time cost increases relatively slightly, while the memory use grows fast.
- IP-TREE/VIP-TREE perform best with relatively less time cost and smaller memory use. For time cost, VIP-TREE always outperforms IP-TREE because of the extra precomputation, but it needs more memory. With the doors increasing, their time and memory costs rise slightly.
- IDMODEL and CINDEX performs worst in both time and memory costs because they have to visit many doors in search.

Table 10: Results of RQ with Decomposition Method

Model	Time (us.)					Memory (MB.)				
	SYN5^0	SYN5	MZB^0	MZB	MZB$^\Delta$	SYN5^0	SYN5	MZB^0	MZB	MZB$^\Delta$
IDMODEL	9695	14999	24065	23527	18917	13	3	12	7	5
IDINDEX	460	704	471	439	349	414	437	815	841	1855
CINDEX	11283	15859	21840	21351	20267	12	4	11	8	5
IP-TREE	8923	123076	7957	17215	26110	88	92	61	58	76
VIP-TREE	6808	57988	4476	11079	19181	111	150	62	59	78

Table 11: Results of kNNQ with Decomposition Method

Model	Time (us.)					Memory (MB.)				
	SYN5^0	SYN5	MZB^0	MZB	MZB$^\Delta$	SYN5^0	SYN5	MZB^0	MZB	MZB$^\Delta$
IDMODEL	4773	9240	14318	14224	12828	9	3	12	6	4
IDINDEX	143	160	180	185	197	461	457	679	796	974
CINDEX	4907	9294	13115	13328	13225	16	4	11	8	6
IP-TREE	7272	33693	3904	7315	10369	112	114	36	36	52
VIP-TREE	6877	24522	3556	5207	7502	117	139	43	55	59

B7 Effect of Decomposition Methods for Hallways

RQ, kNNQ and SPDQ: For RQ and kNNQ, their time cost and memory use with respect to different decomposition methods are reported in Tables 10 and 11. For SPDQ, its time cost, memory use and NVD are reported in Table 12.

- IDINDEX runs fastest when processing RQ and kNNQ but uses most memory. When hallways are decomposed into more partitions, IDINDEX's time cost keeps nearly stable but its memory cost increases. This is because there are more doors connecting increased numbers of partitions, which leads to more door-to-door pairs stored in the distance matrix.
- IDMODEL and CINDEX use the least memory but runs slowest. With more partitions, both time cost and memory use decrease because hallways are decomposed into more partitions each having less doors to process.
- IP-TREE and VIP-TREE perform best considering both time cost and memory use. However, when hallways are decomposed into more partitions, the two methods need more time and memory to process RQ and kNNQ. Regarding the performance in RQ, IP-TREE and VIP-TREE cost more time than IDMODEL. There are more nodes in IP-TREE and VIP-TREE when hallways are decomposed into more partitions, which entails more on-the-fly computations to prune tree nodes when processing RQ and kNNQ. Moreover, the time cost of IP-TREE and VIP-TREE rises faster when processing RQ and kNNQ than processing SPDQ. That is because there is some extra cost to prune nodes when processing RQ and kNNQ. As the nodes increase, this extra cost increases fast.

6.3 Summary of Findings

We summarize all five model/indexes' performance in Table 13 where more stars imply a better performance (lower cost). IDMODEL incurs minimum time and space costs in construction. It

Table 12: Results of SPDQ with Decomposition Method

Model	Time (us.)					Memory (MB.)					NVD				
	SYN5^0	SYN5	MZB^0	MZB	MZB$^\Delta$	SYN5^0	SYN5	MZB^0	MZB	MZB$^\Delta$	SYN5^0	SYN5	MZB^0	MZB	MZB$^\Delta$
IDMODEL	31242	31855	36220	33503	32396	44	63	54	73	92	82574	10074	24877	12718	4243
IDINDEX	138	75	71	69	73	388	396	1096	1273	1489	58	8	22	9	8
CINDEX	32823	26900	35307	33238	31806	43	65	52	97	110	82574	10074	24877	12718	4243
IP-TREE	1610	7523	1252	1893	3257	59	44	31	39	44	416	843	97	87	139
VIP-TREE	856	2379	1091	1126	1474	64	42	54	39	55	136	61	37	24	26

works well for RQ and kNNQ, and its performance for SPQ/SDQ even improves when hallways are decomposed into more partitions. IDINDEX runs fastest for all types of indoor spatial queries while requiring significantly large time to construct offline and high memory consumptions during search. CINDEX performs only comparably to IDMODEL when processing the queries. IP-TREE and VIP-TREE are optimized for SPQ/SDQ tasks—they stand out when there are many C-Pars connected by so-called access doors; they decline when decomposition reduces C-Pars.

In short, IDINDEX is preferred for small-scale spaces. VIP-TREE is recommended if routing is the task or the space accommodates many C-Pars. Otherwise, IDMODEL is recommended for non-routing queries due to its low construction cost and good balance between storage and query time costs.

Table 13: Summary of Findings

Model	Construction Cost		RQ/kNNQ Search		SPQ/SDQ Search	
	Model Size	Time	Memory	Time	Memory	Time
IDMODEL	★★★★★	★★★★★	★★★★★	★★★	★★★★★	★
IDINDEX	★	★	★	★★★★★	★	★★★★★
CINDEX	★★★★	★★★★	★★★★★	★★★	★★★★★	★
IP-TREE	★★★	★★★	★★★★	★	★★★★	★★★
VIP-TREE	★★	★★★	★★★	★★	★★★	★★★★

7 CONCLUSION AND FUTURE WORK

This work reports on an extensive experimental evaluation of five indoor space model/indexes that support four typical indoor spatial queries. Our evaluation concerns the costs in model/index construction and query processing using a model/index. By analyzing the results, we summarize the pros and cons of all techniques and suggest the best choice for typical scenarios.

For future work, changes to existing methods may improve their performance. First, heuristics like A* and IDA* algorithms can replace the Dijkstra-based expansion in IDMODEL and CINDEX to speed up SPDQ processing. Second, intra-partition indexes like grids can be combined with CINDEX and IP-TREE/VIP-TREE to achieve local object pruning in processing RQ and kNNQ. Third, strategies to select crucial doors/partitions can be developed to reduce the storage of door-to-door distances in CINDEX and IP-TREE/VIP-TREE while preserving their search efficiency.

Acknowledgement. This work was supported by IRFD (No. 8022-00366B), ARC (No. FT180100140 and DP180103411), the Key R&D Program (Zhejiang, China) (No. 2021C009) and NSFC (No. 62050099).

REFERENCES

[1] https://github.com/indoorLBS/ISQEA.
[2] Tanvir Ahmed, Torben Bach Pedersen, and Hua Lu. 2014. Finding dense locations in indoor tracking data. In *MDM*. 189–194.
[3] Tanvir Ahmed, Torben Bach Pedersen, and Hua Lu. 2017. Finding dense locations in symbolic indoor tracking data: modeling, indexing, and processing. *GeoInformatica* 21, 1, 119–150.
[4] S. Alamri, D. Taniar, and M. Safar. 2012. Indexing Moving Objects in Indoor Cellular Space. In *NBiS*. 38–44.
[5] Douglas G Altman and J Martin Bland. 1994. Statistics notes: quartiles, quintiles, centiles, and other quantiles. *BMJ* 309, 6960, 996–996.
[6] Anahid Basiri, Elena Simona Lohan, Terry Moore, and et al. 2017. Indoor location based services challenges, requirements and usability of current solutions. *Computer Science Review* 24, 1–12.
[7] Thomas Becker, Claus Nagel, and Thomas H Kolbe. 2009. A multilayered space-event model for navigation in indoor spaces. In *3D geo-information sciences*. 61–77.
[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: an efficient and robust access method for points and rectangles. 322–331.
[9] Muhammad Aamir Cheema. 2018. Indoor location-based services: challenges and opportunities. *SIGSPATIAL Special* 10, 2, 10–17.
[10] Lisi Chen, Gao Cong, Christian S Jensen, and Dingming Wu. 2013. Spatial keyword query processing: an experimental evaluation. *PVLDB* 6, 3, 217–228.
[11] Constantinos Costa, Xiaoyu Ge, and Panos Chrysanthis. 2019. CAPRIO: Context-Aware Path Recommendation Exploiting Indoor and Outdoor Information. In *MDM*. 431–436.
[12] Matthias Delafontaine, Mathias Versichele, Tijs Neutens, and Nico Van de Weghe. 2012. Analysing spatiotemporal sequences in Bluetooth tracking data. *Applied Geography* 34, 659–668.
[13] Zijin Feng, Tiantian Liu, Huan Li, Hua Lu, Lidan Shou, and Jianliang Xu. 2020. Indoor Top-k Keyword-aware Routing Query. In *ICDE*. 1213–1224.
[14] Marcus Goetz and Alexander Zipf. 2011. Formal definition of a user-adaptive and length-optimal routing graph for complex indoor environments. *Geo-Spatial Information Science* 14, 2, 119–128.
[15] Christian S Jensen, Hua Lu, and Bin Yang. 2009. Indexing the trajectories of moving objects in symbolic indoor space. In *SSTD*. 208–227.
[16] Xinlong Jiang, Yunbing Xing, Yiqiang Chen, Yang Gu, and Junfa Liu. 2019. Indoor Trajectory Restoration Method Based on PoI Relation Constraints. In *SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI*. 439–446.
[17] Peiquan Jin, Tong Cui, Qian Wang, and Christian S Jensen. 2016. Effective similarity search on indoor moving-object trajectories. In *DASFAA*. 181–197.
[18] Donald B Johnson. 1973. A note on Dijkstra's shortest path algorithm. *J. ACM* 20, 3 (1973), 385–388.
[19] YongHee Kim, HaRim Jung, JaeHee Jang, and Ung-Mo Kim. 2016. An efficient grid index for moving objects in indoor environments. In *IMCOM*. 1–4.
[20] Jiyeong Lee. 2004. A spatial access-oriented implementation of a 3-D GIS topological data model for urban entities. *GeoInformatica* 8, 3, 237–264.
[21] Huan Li, Hua Lu, Muhammad Aamir Cheema, Lidan Shou, and Gang Chen. 2020. Indoor mobility semantics annotation using coupled conditional Markov networks. In *ICDE*. 1441–1452.
[22] Huan Li, Hua Lu, Lidan Shou, Gang Chen, and Ke Chen. 2018. In search of indoor dense regions: An approach using indoor positioning data. *IEEE Trans. Knowl. Data Eng.* 30, 8, 1481–1495.
[23] Huan Li, Hua Lu, Lidan Shou, Gang Chen, and Ke Chen. 2019. Finding most popular indoor semantic locations using uncertain mobility data. *IEEE Trans. Knowl. Data Eng.* 31, 11, 2108–2123.
[24] Hui Lin, Ling Peng, Si Chen, Tianyue Liu, and Tianhe Chi. 2016. Indexing for moving objects in multi-floor indoor spaces that supports complex semantic queries. *ISPRS International Journal of Geo-Information* 5, 10, 176.
[25] Tiantian Liu, Zijin Feng, Huan Li, Hua Lu, Muhammad Aamir Cheema, Hong Cheng, and Jianliang Xu. 2020. Shortest Path Queries for Indoor Venues with Temporal Variations. In *ICDE*. 2014–2017.
[26] Tiantian Liu, Huan Li, Hua Lu, Muhammad Aamir Cheema, and Lidan Shou. 2020. An Experimental Analysis of Indoor Spatial Queries: Modeling, Indexing, and Processing. arXiv:2010.03910
[27] Hua Lu, Xin Cao, and Christian S Jensen. 2012. A foundation for efficient indoor distance-aware query processing. In *ICDE*. 438–449.
[28] Hua Lu, Chenjuan Guo, Bin Yang, and Christian S Jensen. 2016. Finding Frequently Visited Indoor POIs Using Symbolic Indoor Tracking Data. In *EDBT*. 449–460.
[29] Hua Lu, Bin Yang, and Christian S Jensen. 2011. Spatio-temporal joins on symbolic indoor tracking data. In *ICDE*. 816–827.
[30] Chaluka Salgado. 2018. Keyword-aware skyline routes search in indoor venues. In *SIGSPATIAL-ISA*. 25–31.
[31] Zhou Shao, Muhammad Aamir Cheema, David Taniar, and Hua Lu. 2016. Vip-tree: an effective index for indoor spatial queries. *PVLDB* 10, 4, 325–336.
[32] Zhou Shao, Muhammad Aamir Cheema, David Taniar, Hua Lu, and Shiyu Yang. 2020. Efficiently Processing Spatial and Keyword Queries in Indoor Venues. *IEEE Trans. Knowl. Data Eng.*.
[33] Nico Sun, Erfu Yang, Jonathan Corney, and Yi Chen. 2019. Semantic path planning for indoor navigation and household tasks. In *TAROS*. 191–201.
[34] Zhifeng Wang, Heng Xie, Zeqin Lin, Tao Wen, Chenglong Guo, and Haichu Chen. 2020. The Robot Path Planning Algorithm In Indoor Environment. In *IECON*. 5350–5355.
[35] Emily Whiting, Jonathan Battat, and Seth Teller. 2007. Topology of urban environments. In *CAADFutures*. 114–128.
[36] Michael Worboys. 2011. Modeling indoor space. In *SIGSPATIAL Workshop on Indoor Spatial Awareness*. 1–6.
[37] Xike Xie, Hua Lu, and Torben Bach Pedersen. 2013. Efficient distance-aware query evaluation on indoor moving objects. In *ICDE*. 434–445.
[38] Xike Xie, Hua Lu, and Torben Bach Pedersen. 2014. Distance-aware join for indoor moving objects. *IEEE Trans. Knowl. Data Eng.* 27, 2, 428–442.
[39] Bin Yang, Hua Lu, and Christian S Jensen. 2009. Scalable continuous range monitoring of moving objects in symbolic indoor space. In *CIKM*. 671–680.
[40] Bin Yang, Hua Lu, and Christian S Jensen. 2010. Probabilistic threshold k nearest neighbor queries over moving objects in symbolic indoor space. In *EDBT*. 335–346.
[41] Jiao Yu, Wei-Shinn Ku, Min-Te Sun, and Hua Lu. 2013. An RFID and particle filter-based indoor spatial query evaluation system. In *EDBT*. 263–274.

Structure Detection in Verbose CSV Files

Lan Jiang
lan.jiang@hpi.de
Hasso Plattner Institute
University of Potsdam
Germany

Gerardo Vitagliano
gerardo.vitagliano@hpi.de
Hasso Plattner Institute
University of Potsdam
Germany

Felix Naumann
felix.naumann@hpi.de
Hasso Plattner Institute
University of Potsdam
Germany

ABSTRACT

Numerous data are stored in semi-structured files with ad-hoc layout. Such data are valuable digital assets for various data-driven applications. This work introduces the notion of *verbose CSV files*. Verbose CSV files include content serving different purposes in various positions. They are designed for human visual inspection or statistical report collection. An important preliminary task for extracting information from such files is *structure detection*, in particular classifying lines or cells by their purpose. As manual efforts are infeasible and error-prone for large files or large sets of files, automatic approaches are desirable.

This work addresses both the *line* and the *cell classification* problems on verbose CSV files. Strudel is a supervised learning approach based on a random forest classifier, combined with a set of novel features that fall into three categories: content features, contextual features, and computational features. We annotated five real-world datasets from various domains, on which we tested our approach. Our in-depth experiments show the advantages of Strudel over baseline and state-of-the-art approaches in both line and cell classification tasks.

1 INTRODUCTION

The rapidly growing amount of data promises to be of great value for everyone's day-to-day life, for example, assisting doctors in personalizing healthcare solutions to their patients, scientists conducting open data-based citizen researches, and enterprises making better business decisions. To enable such applications, raw data must be properly processed and analyzed before generating insights. While some data are saved in well-defined formats, such as relational tables or as key-value pairs, that can be readily parsed by dedicated tools, a large quantity of other data are stored in documents with unique structures, for example, CSV files. CSV files are comma-separated values files that provide a great number of data sources for various data-driven tasks, such as data profiling [10, 24], data curation [22, 26, 30], and information extraction [9, 16]. Although there is a standard[1] that stipulates how data shall be stored in CSV files in theory, in practice users and applications do not always conform to it, producing documents with very unique structures.

This work addresses one such type of document: verbose CSV files. Later, we formally define a comma-separated value file as *verbose* if its raw values serve various purposes, such as data, metadata, group headers or notes, and appear in various positions. An example of a real-world verbose CSV file from the "Crime In the US" (CIUS) dataset is given in Figure 1, where groups of cells with different roles are highlighted. This file cannot be directly

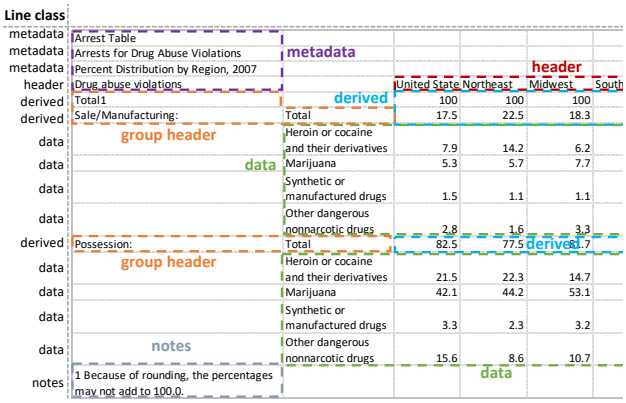


Figure 1: A real-world verbose CSV file with different cell-level and line-level content classes. Here, the line-class is determined by the majority of its cell classes.

ingested by common RDBMS tools, as it contains much additional information, aside from a table with its header and data rows.

While standard CSV files contain data in the form of a structured table, a verbose CSV file is more similar to a spreadsheet, in terms of its flexible content layout. Researchers have suggested that only a minority of spreadsheets (22% of 200 randomly selected spreadsheets) can be directly converted to relational tables [6]. A similar observation by Dong et al. states that less than 3% of spreadsheet tables are "machine-friendly" [11]. Spreadsheets are not the only source for verbose CSV files. We randomly selected 26,140 files from data that we crawled from the Mendeley data portal[2], and manually checked their file types. We grouped all files into three broad categories: (i) application-specific files, such as Microsoft office files; (ii) plain-text files; (iii) multi-media files. Amongst all plain-text files, we are interested in those with at least one table. Out of these files with tabular structures, there are 4,459 files that are verbose, accounting for around 20% of all inspected plain-text files. These verbose plain-text files can be easily transformed to CSV files by applying file-specific delimiters. In our experiments, we tested our approach on verbose CSV files derived from both spreadsheets and arbitrary plain-text files that contain data lines/cells.

Verbose CSV files are prevalent media to store data that aim at aiding human visual inspection or collecting statistical reports. These data often need to be shared amongst communities. Therefore, plain-text files, such as CSV files, are favored due to their generality over those with proprietary file types, such as spreadsheets, that are used by specific applications. However, compared to application-specific files, verbose CSV files are harder to parse, as they do not necessarily follow a specific format. Various open data portals include such files, sometimes labeled as 'ASCII' data[3].

[1] As decribed by RFC 4180: https://tools.ietf.org/html/rfc4180

[2] https://data.mendeley.com

[3] https://data.gov.uk/, https://www.govdata.de/, https://data.europa.eu/euodp/en/data/

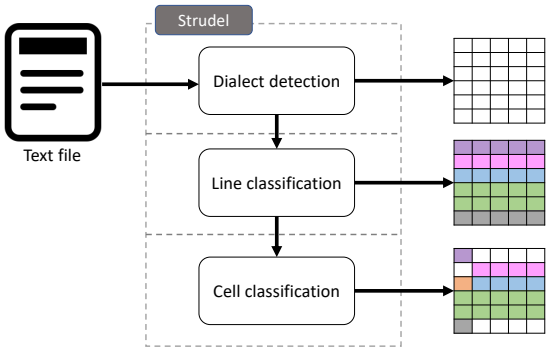


Figure 2: Architecture of the Strudel algorithm.

To obtain insightful information from these files, the first step is to understand their structure, i.e., detect the types of cells, lines, or data blocks. In practice, this is a challenging task due to (i) the wide range of ad-hoc layout variants that can be found and (ii) the lack of rich stylistic features that have been exploited by previous work [2, 11, 18]. Manual inspection to recognize file structure and acquire valuable information is extremely time-consuming for many and large verbose CSV files; automatic methods can support this task.

Information in verbose CSV files is usually organized in a tabular fashion, where each cell is an atomic content unit. File structure is often reflected in the sequence of classes of horizontal lines, as data organized in verbose CSV files usually conforms to the common top-to-bottom data presentation logic. For example, consider the line class labels of the example file in Figure 1. These classes show a natural logic of organizing information: metadata, such as captions, come first, followed by the main body of a table incorporating table headers, derived (aggregated) lines and data lines, and finally a few footnote lines conclude the file. In this work, we aim at *verbose CSV file structure detection* by means of classifying file content in two granularities: *lines* and *cells*.

Obtaining manually labeled datasets with known line and cell annotations to evaluate an approach for the structure detection problem is difficult, as ascertaining these classes in a verbose CSV file is a difficult task even for experienced practitioners. In our study, even by using a tool with a sophisticated graphical interface, practitioners 1) took on average of two minutes to label the lines in a single file, because they needed to spend a lot of time understanding the unique structure of each file; 2) at times disagreed with each other on the annotations of individual lines. These observations highlight the challenge of automatically classifying lines and cells in verbose CSV files.

To address the line/cell classification problem, we propose the Structure Detection in Verbose CSV Files (Strudel) approach, which is grounded on a multi-class random forest classifier. Figure 2 shows the architecture of the approach. It first detects the dialect of a text file, and creates a verbose CSV file from it, based on the dialect. Then Strudel classifies first lines and then cells therein with the proposed feature sets. Cells of different types are distinguished by colors. Sections 4 and 5 describe Strudel for line and cell classification, respectively. We propose sophisticated features to model the individual classes for both classification tasks. The features can be categorized into three groups: 1) *content features* parsing the values of cells or lines, such as cell length and amount of words; 2) *contextual features* comparing the inspected cell or line with its neighbors, such as the similarity of data types between lines/cells; 3) *computational features* seeking to connect lines/cells with each other by inspecting arithmetic correlations between them. The contributions of this work are summarized as follows:

(1) A supervised learning approach with novel features to address the structure detection problem for verbose CSV files.
(2) A dataset with more than 97,000 annotated lines in 226 files, reforged labels of datasets from related work based on our perspective on cell classes, and a dataset with 62 files transformed from plain-text files.
(3) An experimental comparison of Strudel with baseline and state-of-the-art approaches.

In the next section, we introduce the related work on line and cell classification tasks and other relevant areas. In Section 3, we formally define the classification problem and introduce the set of line and cell classes. Section 4 and 5 describe the core idea of Strudel, followed by Section 6, where we present the results and analysis of our experiments. We conclude in Section 7.

2 RELATED WORK

Extracting information from semi-structured documents has been a growing research field in recent years. Relevant research questions include how to locate tabular content in documents such as PDF files [20], and spreadsheets [12], how to distinguish relational tables from non-relational tables [4, 29], and how to extract relational tables from heterogeneous sources [5, 13, 14, 25].

Prior to extracting information from a semi-structured document, understanding its structure is necessary. Some techniques have been proposed to address the structure detection problem on various documents, such as web tables and spreadsheets, which include tabular material and have flexible layout. We summarize these works focusing on structure detection by classifying lines or cells, respectively.

2.1 Line classification

Pinto et al. suggest a conditional random field (CRF) learning approach to predict the label for lines of plain-text documents crawled from a open data portal [23]. For each document, a sequence of features is computed for its lines. The sequences of all documents are used by the CRF classifier to infer the label. This approach was later adopted to infer spreadsheet table schemata [28] and extract relational data from spreadsheets [5]. Moreover, it was extended by Adelfio et al. to recognize line classes in web tables and spreadsheets [2]. The authors suggest feature binning to generalize the training data and show the effectiveness of their approach on recognizing line classes in both HTML tables and spreadsheets crawled from several open data portals. However, the approach assumes the presence of stylistic features, such as font styles, or built-in spreadsheet formula features, which are not available in verbose CSV files.

A recent work has proposed the rule-based approach Pytheas for CSV file line classification [8]. To classify lines in a CSV file, the approach first determines for each line whether it is data or non-data by consulting a set of fuzzy rules, whose weights have been learned beforehand with a training dataset. These binary results are then used to determine the top/bottom borders of tables in the file. Finally, the approach exploits additional class-specific rules on the discovered table/non-table areas to further ascertain the class of each line. The core of this approach is the

design of the fuzzy rule set, which significantly impacts the consequences of table border discovery, and also line classification. However, such a fixed set of rules might fail to generalize to new circumstances in unseen data.

2.2 Cell classification

Finer-grained cell classification in tabular documents has been the subject of academic interest in recent years. Abraham et al. developed the UCheck framework, which includes a component to detect "cell roles", such as header and footer, in spreadsheets using several heuristics [1]. Cell roles are then used by the system to detect spreadsheet errors. The goal of their approach is to correlate cells in a table with their corresponding headers, thus they assume spreadsheets with only table regions as input data.

Koci et al. suggest a supervised learning approach with a post-processing component to repair classification errors [19]. The authors introduce five misclassification patterns and suggest that the occurrence of them in the results hints at a misclassification.

To reduce the amount of manual annotation effort, Chen et al. integrated an active learning technique into their spreadsheet cell classification approach [7]. In their iterative algorithm, a sheet selector presents the most uncertain spreadsheet to human labelers. The sheet is then labeled and included into a training set that is used to train a spreadsheet property classifier.

Ghasemi-Gol et al. suggest a recursive neural network (RNN) architecture on two seperately trained cell embeddings that capture the contextual and the stylistic semantics of cells, respectively [18]. Even though the authors mention the contextual impact on a cell from both neighboring and distant cells, they considered only the former ones in their approach. They built the stylistic features upon those suggested by [19].

In summary, these works all make use of stylistic features of their input. However, no such information can be obtained in verbose CSV files. In this work, we compare our approach with the RNN-based approach of [18], as it was reported to outperform the other approaches. In spite of using stylistics features to solve the task, the authors also reported the performance of their algorithm without their usage, enabling a direct comparison to our approach.

3 DEFINITIONS AND PROBLEM STATEMENT

We first define the notion of verbose CSV files. Then, we present the taxonomy of element classes used to label our datasets and present the detailed definition of each class. Based on these concepts, we formally state the problem addressed by our approach.

3.1 Verbose CSV files

A standard CSV file, according to RFC 4180, contains an optional header line at the beginning of the file, followed by a number of data lines. In contrast to that, a *verbose CSV file* may include elements of heterogeneous classes (which will be introduced in Section 3.2), possibly with empty visual separators. Here, an *element* is either a non-empty cell or a line that includes at least one non-empty cell.

Definition. *A* verbose CSV file *is a comma-separated values file with values including one or more of metadata, header, group, data, derived, and notes at arbitrary positions. Each line of the file may be composed of cells of one or more classes. Empty cells may represent either missing values or serve layout purposes.*

A standard CSV file stores a single table that is machine-readable, whereas a verbose CSV file may include multiple tables, and make use of empty cells and further cell types to improve human readability, leading to various configurations. Information of different kinds may be organized as connected clusters of cells throughout the file: each such cluster may include information of different types, such as data, metadata, or aggregations; a table may be divided by blank visual separators into several table fractions; etc.

3.2 Class taxonomy

Our taxonomy includes six semantic classes and is similar to that of [2], which addressed the line classification problem on web tables and spreadsheets. While in principle content of any class may appear at any location in a verbose CSV file, we enforce a few practical constraints on their possible position, reflecting the usual reading convention: from left to right and from top to bottom, assuming that tables are stacked only vertically. We describe in detail each class in the following list.

- **metadata.** Metadata are the descriptive text *above* a table. Such text may include the title of a table, or additional information on the content of the table. A metadata area may span across one or more lines and columns.
- **group.** In verbose CSV files, tables are often split into several fractions, each including data of a particular group. A group (a.k.a. group header) element serves as the label of such a fraction. We have seen in our datasets the group elements appearing both *above* and *below* header areas. Therefore, we allow both cases in our definition. Group cell may also serve as the leftmost string cells in a derived line, e.g., the 'Sale/Manufacturing:' cell in Figure 1.
- **header.** Headers are the column labels in the top area of a table (or table fraction). Headers may span multiple cells. In our definition, the header elements are located *above* the data area, and *below* any metadata block of the table.
- **data.** Data elements are the content of a table that cannot be derived from any other elements. Because they constitute the main body of a table, data elements of a section of a table are always *below* header and group elements that indicate this section.
- **derived.** A derived cell aggregates the values of some other numeric cells in the same table. In verbose CSV files, derived cells are usually organized as the top- or bottom-most lines, or the left/right-most columns of the data area of a table.
- **notes.** Notes are descriptive text that follow a table. They may give explanations of particular parts of a table, explain the meaning of marks used in the table, or indicate the data source origin.

Any line or cell in a verbose CSV file can be associated with exactly one of the classes introduced above. We address the following problem: *Given a verbose CSV file and the group of possible element classes, how can we determine the classes of all elements?* We consider elements of two natures addressing two sub-problems under the same problem definition: *lines*, reflecting the usual vertical arrangement within a file, and *cells*, as the most fine-grained element of a structured file.

4 LINE CLASSIFICATION

In this section, we describe *Strudel*^{L} – the Strudel approach to deal with the line classification problem. *Strudel*^{L} is based on a

multi-class random forest classifier. Its input includes a set of features extracted from the two-dimensional tabular data.

Various kinds of features have been proposed by previous work to address the line classification problem, including content features, contextual features, spreadsheet formula features, and stylistic features [2]. In our case, formula and stylistic features cannot be applied, as CSV files do not preserve these rich-text information. Instead, we design a set of content features, contextual features, as well as computational features. We build the features of *Strudel*^{L} on top of the applicable features from the previous work [2]. Table 1 lists our complete set of features, divided into the three groups. The context features can include information from both the line above and the line below. Thus, the features marked by a star are applied twice – once for the line above and once for the line below. To distinguish derived cells from data cells, we propose novel *computational* features that check whether the value of a numerical cell can be calculated by applying a specific aggregation function on the numbers in the vicinity, i.e., the cells in the same row or column.

Table 1: Line classification features: '∗' marks contextual features applied to both lines above and below the inspected line; '†' are adapted from [19].

Category	Feature	Value
Content	EmptyCellRatio†	[0.0, 1.0]
	DiscountedCumulativeGain	[0.0, 1.0]
	AggregationWord†	0/1
	WordAmount	[0.0, 1.0]
	NumericalCellRatio†	[0.0, 1.0]
	StringCellRatio†	[0.0, 1.0]
	LinePosition†	[0.0, 1.0]
Contextual	DataTypeMatching*	[0.0, 1.0]
	EmptyNeighboringLines*	[0.0, 1.0]
	CellLengthDifference*	[0.0, 1.0]
Computational	DerivedCoverage	[0.0, 1.0]

Here, we describe and explain only the novel features used in our approach and refer to related work for the others.

- DiscountedCumulativeGain (DCG) calculates the discounted cumulative gain on a vector created from the cells of a line. The vector has the same length as the number of cells in a line. An element is set to '1' if the corresponding cell in the line is non-empty, or '0' if it's empty. This feature is exploited to model the pattern of empty cells. DCG gives more weight to left-more positions than to right-more positions, modeling users laying out data from left to right.
- AggregationWord checks whether a line contains any word that belongs to a pre-made dictionary of terms associated with aggregation in tables (case-insensitive): `total`, `all`, `sum`, `average`, `avg`, `mean`, and `median`. An existence of any keyword gives '1' to this feature, otherwise '0'. Using a dictionary of such kind of keywords proves to be effective [19].
- WordAmount calculates the number of words in all cells of a line. A word is a sequence of alphanumeric characters. The feature values are normalized per file by using a min-max normalization strategy.
- DataTypeMatching calculates the percentage of line cells whose data types match with those of the adjacent line (above or below). Note that some files insert an empty line between every pair of non-empty lines to visually highlight the content. However, comparing the data type of a line with an empty adjacent line does not carry much information. Therefore, an adjacent line refers to the closest non-empty line. Data and derived lines tend to have numerical cells while header lines usually contain alphanumeric values. Other functional lines, such as metadata, notes, and group lines tend to have many empty cells, as they often contain values for only the first cell in a line.
- EmptyNeighboringLines calculates the percentage of empty lines in the five lines above or below the inspected line. Empty lines are often used as visual separators in verbose CSV files. Using such a separator between data lines within a table is uncommon, but placing them between two classes of lines, such as header-data and derived-notes is more common.
- CellLengthDifference calculates the cell value length difference between two adjacent lines by calculating the Bhattacharyya histogram difference on the sequence of cell value length of the two lines. When computing this feature, we compare only a line with its closest non-empty neighboring line, similar to that applied for the DataTypeMatching feature. While data lines tend to have similar cell-wise value lengths, as they usually describe the same property and thus draw values from the same domain and range, non-data lines might have natural language form values that are of arbitrary value lengths.
- DerivedCoverage counts the number of numeric cells that are recognized as derived cells by the derived cell detection Algorithm 2 in the next section. The feature is normalized by the number of numeric cells in this line.

Note that these line classification features are all local features, i.e., describing the characteristics of individual lines. We have tested a few global features that reflect properties of the entire file, namely percentage of empty lines in a file, width and length of a file, and the number of empty line blocks in a file. However, our experiments show no positive impact on the classification problem.

All features are normalized and passed to a random forest classifier that predicts one class for each line. When used as the Line class probability feature in *Strudel*^{C} (Section 5.4), the output is a set of vectors, each of which stands for a probability vector of all classes for a line.

5 CELL CLASSIFICATION

Our cell classification approach *Strudel*^{C} is, like *Strudel*^{L}, based on a multi-class random forest classifier. For the input of this classification task, we have again constructed a set of features that include both the effective ones from previous work, and novel ones. The predictions for line classes are used as a set of features in *Strudel*^{C}. Therefore, the *Strudel*^{L} approach is executed beforehand to obtain the line prediction probabilities that are then transformed into the features of *Strudel*^{C}. We leave the detailed description to Section 5.4.

5.1 Feature extraction

Previous work has proven the effectiveness of content features, stylistic features, spreadsheet formula features, and contextual features [18, 19]. We ignore spreadsheet-specific formula features and stylistic features, as they cannot be constructed from verbose CSV files. Table 2 lists all features involved in our approach, which fall into three groups: content, contextual, and computational features.

Table 2: Cell classification features; '∗' marks contextual features applied to each of the eight surrounding cells of the inspected cell; '†' marks features from related work.

Category	Feature	Value
Content	ValueLength†	[0.0, 1.0]
	DataType†	[0..4]
	HasDerivedKeywords†	0/1
	RowHasDerivedKeywords†	0/1
	ColumnHasDerivedKeywords†	0/1
	RowPosition†	[0.0, 1.0]
	ColumnPosition†	[0.0, 1.0]
	LineClassProbability	$(p_1, ..., p_6)$
Contextual	IsEmptyRowBefore	0/1
	IsEmptyRowAfter	0/1
	IsEmptyColumnLeft	0/1
	IsEmptyColumnRight	0/1
	RowEmptyCellRatio†	[0.0, 1.0]
	ColumnEmptyCellRatio†	[0.0, 1.0]
	BlockSize	[0.0, 1.0]
	NeighborValueLength*	[0.0, 1.0]
	NeighborDataType*	[0..5]
Computational	IsAggregation	0/1

The features marked with '†' in Table 2 are based on those used in [18, 19]. Some of the original features are integrated in our feature set without modification, such as ValueLength and DataType, while others are adapted to a certain extent. For instance, a Boolean feature used to mark the existence of derived cell keywords is extended to a row or a column (RowHasDerivedKeywords and ColumnHasDerivedKeywords), i.e., whether the row or the column that contains the inspected cell contains any derived cell keywords. Features without '†' are new. ValueLength counts the number of characters in the value of the cell. DataType in this work has four possible values, corresponding to four data types: int, float, string, and date. In the next subsections, we explain the intuition and implementation of the four most sophisticated of them.

5.2 Block size

A verbose CSV file may contain multiple tables in various positions, rather than a single relational table. Apart from tables, a verbose CSV files may contain non-data regions composed of aggregation cells, notes, or metadata cells. In our datasets, non-data regions are usually smaller than tables.

To model this phenomenon, we create for each non-empty cell a BlockSize feature, which is calculated as the size of the connected component that contains this cell. A connected component is composed of a group of connected, non-empty cells. Two cells are connected if they are either vertically or horizontally adjacent to each other, or there is at least one connective path between them. Algorithm 1 describes how the value of this feature is produced for each cell in a given verbose CSV file. It takes all non-empty cells in a table as input, and outputs key-value pairs where keys are these non-empty cells and values are their respective block sizes. To obtain the block size for all non-empty cells, the algorithm employs a depth-first strategy to iterate over all of them in a given file. It starts from a single cell block (line 4-7), and continuously adds adjacent cells to expand the block until no more non-empty adjacent cell can be found (line 8-13). The block size is normalized to $[0, 1]$ by the size of the file (line 14). The algorithm terminates once all cells have been touched. Regarding the complexity of this algorithm, assume there are n non-empty cells in a verbose CSV file. On the one hand, each cell will be visited once and only once, resulting a $O(n)$ complexity. On the other hand, the four directions of a cell are checked once the cell is visited, leading to a $O(4n)$ complexity. Therefore, the overall algorithm complexity is $O(n) + O(4n) = O(n)$.

Algorithm 1: Block size calculation

Input: The set of non-empty cells in a table C
Output: The set of key-value pairs BS from cells to block size

1 $BS \leftarrow \{\}$;
2 $V \leftarrow \{\}$ # *visited cells* ;
3 **while** $C - V \neq \emptyset$ **do**
4 $\quad c \leftarrow$ random cell in $C - V$;
5 $\quad bs \leftarrow 1$;
6 $\quad V \leftarrow V \cup c$;
7 $\quad B \leftarrow \{c\}$;
8 $\quad$ **while** *there exist cells in* $C - V$ *adjacent to* B **do**
9 $\quad\quad c_{adj} \leftarrow$ an adjacent cell in C;
10 $\quad\quad bs \leftarrow bs + 1$;
11 $\quad\quad V \leftarrow V \cup c_{adj}$;
12 $\quad\quad B \leftarrow B \cup \{c_{adj}\}$;
13 $\quad$ **end**
14 $\quad bs \leftarrow$ normalize(bs);
15 $\quad$ **foreach** $c \in B$ **do**
16 $\quad\quad BS \leftarrow BS \cup \{c : bs\}$;
17 $\quad$ **end**
18 **end**
19 **return** BS

5.3 Neighbor profile

Cells of some classes may be likely to have particular kinds of neighboring cells. For example, to highlight group header cells, users often separate them from other cells with empty cells, or they place derived cells at the margin of a table, as a way to summarize data. These observations bring our focus on the adjacency context of a cell: for each cell, we gather the data types and value lengths of all eight surrounding cells and present each as a single feature in the feature vector. The neighbor profile of a cell includes all these NeighborValueLength and NeighborDataType features. For the cells on the margins of a file, some adjacent cells do not exist. We set a default value for these non-existent adjacent cells, i.e., -1 for value length and data type.

5.4 Line class probability

Despite the possible flexible layout, verbose CSV files are usually organized in some structurally meaningful way. Lines tend to organize mostly homogeneous types of cells to ease human understanding. For example, a data line contains mostly data cells, while a header line contains mostly header cells. Table 3 displays statistics about the *cell class diversity degree* of all lines in our datasets. The cell class diversity degree of a line is its number of distinct non-empty cell classes. We observe that most lines a have diversity degree of 1: for the cells in these lines, their classes are trivially determined by the class of the line. Therefore, when determining the class of a cell, the class of the line it is located in

is likely a useful feature. In fact, we use this feature alone as one of our baselines.

Table 3: Percentage of lines under different diversity degrees.

	Diversity degree				
Dataset	1	2	3	4	5
SAUS	86.3%	13.7%	0%	0%	0%
CIUS	88.7%	11.2%	0.1%	0%	0%
DeEx	95.3%	4.6%	0.1%	0%	0%

To obtain the line class information, we first run $Strudel^L$ to obtain the prediction for each line. The result of this execution is, however, a probability vector of all classes, instead of a single predicted class. We interpret this probability vector as the classifier's confidence for these classes. Each element of the 6-dimensional vector accounts for a feature for the cell class detection.

5.5 Derived cell detection

If a cell is indeed a derived cell, it should be possible to derive its value by aggregating values of some other neighboring cells. This fact has not been considered by previous work, possibly due to computational cost. We propose a derived cell detection algorithm that seeks to identify derived cells by arithmetically correlating their values with other numeric cells.

We made three observations while investigating the datasets: (i) a derived cell usually aggregates the values of cells from its own row or column; (ii) a derived cell tends to aggregate values close to it; (iii) sum and mean are the two dominant aggregation functions used in verbose CSV files. We integrate these insights into our Algorithm 2. For conciseness, it shows only the approach for detecting derived sum cells.

The algorithm takes as input a table as a two dimensional array, the derived keyword dictionary to look for anchoring cells, an aggregation delta d to give some slack to aggregation results, and a coverage threshold c that controls the generality of aggregation results. Executing the algorithm produces all detected derived cells.

We first determine derived cell candidates, as calculating all aggregation possibilities for all numeric cells is prohibitively expensive. We found that some indicative words usually appear in a cell in the same row or column where there exist many derived cells. For example, for a row with many summing cells, words such as 'Total' are likely to appear in a cell of this row. Therefore, we mark those cells with any of our aggregation keywords (introduced in Section 4) as anchoring cells (line 2). Based on our first observation, only numeric cells in the same row or column as an anchoring cell are treated as derived cell candidates (line 6-8).

For the candidates in the same row as the anchoring cell, the algorithm looks first upwards and then downwards for possible aggregating relationships (line 9-19), whereas for the candidates in the same column as the anchoring cell, the algorithm looks left or right (line 20-30). When looking upwards, the algorithm adds numeric values of a row each time to the sum vector correspondingly, and inspects whether the current sum vector is element-wise close enough (according to d) to the candidates. If the coverage of the close enough elements in the sum vector surpasses c, the candidate is treated as a derived cell (line 14-17). Due to our second observation, a row closer to the row where the candidates are is inspected earlier than a row farther away.

Algorithm 2: Derived cell detection

Input: Table T, keywords K, aggregation delta d, coverage c
Output: All detected derived cell C_D

1 $C_D \leftarrow \{\}$;
2 $A \leftarrow$getAnchoringCells(T, K);
3 **if** *A is empty* **then**
4 **return** C_D;
5 **foreach** *a in A* **do**
6 $i_a, j_a \leftarrow$ row index of a, column index of a;
7 $C_R, cc_{ind} \leftarrow$ the list of numeric cells in row i_a and their column indices;
8 $C_C, rc_{ind} \leftarrow$ the list of numeric cells in column j_a and their row indices;
  /* line 9-19 for upwards detection */
9 $sum \leftarrow (0...0)$;
10 **for** $i = 1$ **to** ∞ **do**
11 **if** $i_a - i < 0$ **then**
12 **break**;
13 **else**
14 $v_o \leftarrow$ numeric values at cc_{ind} in row $(i_a - i)$;
15 $sum \leftarrow sum + v_o$;
16 **if** *coverage of* $(C_R - sum < d) > c$ **then**
17 $C_D \leftarrow C_D \cup C_R$
18 **end**
19 **end**
  /* repeat line 9-19 for downwards detection */
  /* line 20-30 for leftwards detection */
20 $sum \leftarrow (0...0)$;
21 **for** $i = 1$ **to** ∞ **do**
22 **if** $j_a - i < 0$ **then**
23 **break**;
24 **else**
25 $v_o \leftarrow$ numeric values at rc_{ind} in column $(j_a - i)$;
26 $sum \leftarrow sum + v_o$;
27 **if** *coverage of* $(C_C - sum < d) > c$ **then**
28 $C_D \leftarrow C_D \cup C_C$
29 **end**
30 **end**
  /* repeat line 20-30 for rightwards detection */
31 **end**
32 **return** C_D

6 EVALUATION

In this section, we first describe the datasets and all algorithms used in our experiments. After that, we present our experimental evaluation on Strudel, including its comparison with referenced approaches, analysis of feature importance and four advanced features on cell classification, and difficult case study for both line and cell classification tasks.

6.1 Datasets and experimental setup

This section first lists the datasets used in our evaluations, and the preprocessing steps applied thereon. In practice, verbose CSV files may have unique dialects. The dialect of a file specifies the delimiter, quoting character, and escape character, enabling to parse the lines and cells correctly. Therefore, as a general preprocessing, we first applied dialect detection on each file with the approach of van den Burg et al. [27]. This approach takes a text file as input, and produces its detected dialect. The scope of each cell or line is determined by that dialect. The second part of this section describes the list of baseline and competing approaches,

our Strudel approach, and their respective configurations for evaluation.

6.1.1 Datasets. Our datasets with verbose CSV files come from various sources, as summarized in Table 4. Only non-empty lines and cells are counted. The content of our datasets is given in the English language and follows a top-bottom / left-right organisation. Non-Western verbose CSV files might organize the content in a different fashion, which could be an interesting future work.

We created the dataset *GovUK* by crawling all data files in Microsoft Excel format (both in .xls and .xlsx) from an open data portal[4] and transforming them into corresponding CSV format with the Apache POI library[5]. While converting them to CSV files, we omitted both files that contain macros or were not otherwise processable by the library and empty sheets. We randomly selected a subset of 226 files from the dataset, and created a line-level ground truth for them. To perform the actual annotation, we implemented a tool to annotate each line as one of our element classes. Each line of each file in the created dataset was annotated by three human experts. In case of disagreement, which affected only 1% of the annotated lines, we used majority-vote to determine the annotation. For the lines with complete disagreement (fewer than 250 lines in our dataset), we employed an independent fourth annotator to determine which one of the three answers to apply. In the end, we obtained the ground truth for in total more than 110,000 annotated lines. We make all datasets publicly available[6].

Table 4: The dataset overview.

Dataset	# files	# lines	# cells
GovUK	226	97,212	1,382,704
SAUS	223	11,598	157,767
CIUS	269	34,556	367,172
DeEx	444	77,852	784,229
Mendeley	62	195,598	1,359,810
Troy	200	4,348	23,077

Three other datasets *SAUS*, *CIUS*, and *DeEx* were created and annotated by Ghasemi-Go et al. for cell classification [18]. The first two are administrative datasets, while the last one is a business dataset. More detailed descriptions of each dataset can be found in their original paper. The datasets were annotated by the original authors with a slightly different taxonomy. To reconcile their annotations to ours, we partly re-annotate their labels. In summary, they annotated all left-most headers of a table as *attributes*, while we consider them as data of their columns. In the example of Figure 1, they treat the cells that indicate the drug types in the second column as *attributes*, while we annotate them as *data*, as we model them as a data column of the table without a header. We also note that some clearly derived cells were marked as data: understandable errors due to their similarity, which we corrected. In many cases, derived cells form an entire line, with the exception of the leading cell, which is usually textual. This textual cell often includes keywords, such as 'Total', and is neither a derived cell nor a header cell. We treat it as *group* in our system, because a derived line often serves as a section separator in a table. Table 5 presents the class distribution of these three datasets with the reforged annotations.

Table 5: The number of lines or cells per class in the dataset *SAUS*, *CIUS*, and *DeEx* as a whole.

class	# lines	# cells	# cells per line
metadata	2,213	2,479	1.12
header	2,232	19,047	8.53
group	1,767	6,143	3.48
data	114,354	1,202,058	10.51
derived	1,406	76,996	54.76
notes	2,036	2,445	1.20
Overall	124,006	1,309,168	-

Our *Mendeley* data is a set of plain-text files collected from Mendeley's data sharing platform[7] of experimental data. These data are stored in research projects. We crawled all 2,214 projects whose data are stored on Mendeley's own server and that contain at least one plain-text file, i.e., whose MIME type is "text/*". This MIME type corresponds to a wide variety of actual file formats: not only files with table structures and verbose information, such as verbose CSV files, but also programming scripts, HTML pages, etc. We randomly selected 100 projects that include at least one suitable verbose text file, and obtain one such file from each project. Given the intricate dialects of these plain-text files, the dialect detection approach of [27] cannot reliably discover the correct dialect for all files. A file is *parse-able* if the dialect for the table region (including header, data, group, and derived) is correct. For our experiments, we kept the 62 parse-able verbose CSV files. Note that this dataset is used only to verify the performance of our approach on verbose plain-text files and is not part of. the training set. We observe a high line-to-file and cell-to-file ratio, because the files of this dataset are mostly used to store data, e.g., experimental results, rather than presenting statistical tables.

The last dataset, *Troy*, contains 200 CSV files collected from various government websites [17]. Embley et al. used this dataset in their work to convert different statistical tables to relational tuples [15]. We kept the dataset unseen during the design of Strudel to test the out-of-domain generalizability of our approach with this dataset.

In our data preparation process, we cropped each file by removing the marginal empty lines or columns, as some of our features are sensitive to the number of empty cells in the lines, and leading/trailing empty lines are trivial cases. Values of spanning cells in original spreadsheets are copied only to the top-left cell in the CSV file, instead of to all covered cells for two reasons: (i) the top-left is well-defined for all shapes of spanning cells and (ii) copying the values to all covered cells creates too many repeated characters, confusing the models that cause unnecessary over-fittings towards these values. To ease future study on this topic, we will publish all datasets and their annotations.

6.1.2 Setup of experiments. The list below contains all algorithms used in the evaluation, along with their corresponding configurations. All algorithms were implemented in Python with the scikit-learn library[8]. The superscript in the name of an algorithm indicates the type of elements detected by this algorithm, i.e., 'L' and 'C' represent line and cell classification, respectively.

[4] https://data.gov.uk/
[5] http://poi.apache.org/index.html
[6] https://hpi.de/naumann/projects/data-preparation.html
[7] https://data.mendeley.com/, last crawled on 3. August 2020
[8] https://scikit-learn.org/stable/index.html

Table 6: Per-class and overall F-1 score on each dataset for line classification (top) and cell classification (bottom).

		metadata	header	group	data	derived	notes	accuracy	macro-avg
GovUK	CRF^L	**.789**	.379	.898	**.991**	.339	.752	**.979**	.733
	$Pytheas^L$	.446	.444	.172	.986	-	.545	.970	.518
	$Strudel^L$	.670	**.774**	**.919**	.989	**.361**	**.797**	.978	**.751**
	# lines	*878*	*519*	*850*	*93,584*	*665*	*716*	-	-
SAUS	CRF^L	.893	.651	.817	.963	.477	.980	.931	.797
	$Pytheas^L$	.884	.768	.741	.973	-	.814	.944	.836
	$Strudel^L$	**.984**	**.960**	**.882**	**.987**	**.599**	**.984**	**.976**	**.899**
	# lines	*469*	*565*	*289*	*9,346*	*279*	*650*	-	-
CIUS	CRF^L	**.994**	.961	**.992**	**.996**	.749	**.988**	.992	.947
	$Pytheas^L$	.988	.867	.000	.970	-	.637	.943	.692
	$Strudel^L$	**.994**	**.972**	.984	**.996**	**.834**	.978	**.993**	**.960**
	# lines	*1,034*	*435*	*1,074*	*30,890*	*449*	*674*	-	-
DeEx	CRF^L	.753	.373	.027	.970	.244	.480	.942	.475
	$Pytheas^L$	.564	.406	.137	.980	-	.433	.957	.420
	$Strudel^L$	**.797**	**.807**	**.357**	**.989**	**.548**	**.761**	**.976**	**.710**
	# lines	*710*	*1,299*	*407*	*74,116*	*678*	*712*	-	-

		metadata	header	group	data	derived	notes	accuracy	macro-avg
SAUS	$Line^C$	.963	.915	.451	.970	.332	.888	.930	.753
	RNN^C	.977	.925	.466	.956	.345	.902	.919	.762
	$Strudel^C$	**.987**	**.972**	**.752**	**.983**	**.689**	**.957**	**.968**	**.890**
	# cells	*469*	*4,769*	*825*	*142,301*	*8,708*	*695*	-	-
CIUS	$Line^C$	.991	.973	.361	.929	.156	.937	.824	.725
	RNN^C	.987	.976	.679	.904	.443	.963	.850	.825
	$Strudel^C$	**.993**	**.993**	**.916**	**.946**	**.465**	**.989**	**.895**	**.884**
	# cells	*1,035*	*3,838*	*4,228*	*310,354*	*47,043*	*674*	-	-
DeEx	$Line^C$	.630	.625	.155	.981	.258	.520	.955	.528
	RNN^C	.623	.772	.347	.952	.244	.413	.930	.559
	$Strudel^C$	**.689**	**.801**	**.444**	**.988**	**.683**	**.598**	**.977**	**.700**
	# cells	*975*	*10,314*	*1,216*	*749,403*	*21,245*	*1,076*	-	-

- CRF^L is a conditional random field-based learning approach dedicated to line classification from the work of Adelfio et al. [2] as the current state of the art. We applied this approach with the logarithmic binning technique introduced by the authors, as this setting was reported to gain the best performance.
- $Pytheas^L$ is a rule-based approach that discover the locations of tables, and further classifies the lines in CSV files [8]. We use the parameter values introduced in the original paper for our experiments.
- $Strudel^L$ is our proposed approach for line classification. The underlying random forest classifier used the default settings in the scikit-learn library.
- $Line^C$ is a baseline approach for cell classification. This approach simply extends the predicted class of a line from the result of a $Strudel^L$ execution to each non-empty cell in this line.
- RNN^C is based on the state of the art by Ghasemi-Gol et al., which classifies cell types with a recursive neural network using pre-trained cell embeddings [18]. For our experiment, we used the same settings as introduced in the original paper.
- $Strudel^C$ is our approach for cell classification. Again, we used the default settings of the random forest classifier in the scikit-learn library. In our experiment, we do not observe a substantial difference in the result with different values of the aggregation delta d and coverage c. We set them to 0.1 and 0.5, respectively.

Apart from using content and spatial features, both original CRF^L and RNN^C applied stylistic or spreadsheet formula features. Because the input data in our use-case are style-less, verbose CSV files, we remove all stylistic features from the two approaches so as to conduct fair comparisons. Each algorithm is evaluated using 10-fold cross validation. When creating the folds, our process ensures that all elements from a single file appear in either the training or the test set. We repeat the 10-fold cross validation ten times to reduce bias leaning to particular fold splits. The results of all repetitions are averaged to obtain the final score.

We have tested several classification algorithms for Strudel, including Naïve Bayes, KNN, SVM, and random forest. Random forest consistently outperformed the other candidate algorithms on our datasets for both classification tasks. Therefore, we chose it as the backbone supervised learning algorithm of Strudel. The advantage of random forest over the other algorithms is that it reduces the risk of over-fitting by considering the results from multiple base classifiers, which is crucial for unbalanced datasets, such as verbose CSV files.

6.2 Comparative evaluation

This section presents the comparative evaluation results between Strudel and the referenced approaches. We use the F1 measure to evaluate the classification correctness of each approach for both line and cell classification tasks. When comparing the overall result amongst algorithms, we focus on the macro average, which does not weigh the average score with the support of individual classes, thus avoiding the bias from the number of

per-class instances, which is crucial for supervised learning tasks on imbalanced data.

6.2.1 Line classification. We compared *Strudel*^{L} with *CRF*^{L} and *Pytheas*^{L}. *CRF*^{L} uses a set of features, including content features, contextual features, and stylistic features to train a conditional random field based classifier on web tables and spreadsheets. *Pytheas*^{L} uses a number of weighted rules to decide whether a line is data or non-data. The binary results are used to draw the table top/bottom boundaries, on top of which the approach utilizes some additional rules to determine line classes.

Table 6 (top) reports the per-class and macro-average F1 scores, and accuracy for the three approaches. Note that *Pytheas*^{L} can classify a line as one of only five classes that correspond respectively to ours, missing the derived class. Therefore, when calculating the measurements for this approach, we leave out the derived lines from our datasets. Overall, our approach leads on macro-average for all datasets. *Pytheas*^{L} does not perform well in general on the minority classes in all but the *SAUS* datasets, as its proposed rules are not suitable for these datasets: they produce poor results already for the binary data/non-data classification, which disrupts the subsequent table discovery and line classification. Group lines are particularly difficult for *Pytheas*^{L}: the scope of group lines is constrained to lines between data lines and has only the leftmost cell non-empty. While the group lines in *SAUS* mostly follow this definition, those in the other datasets do not. Most header lines of both *SAUS* and *CIUS* are across few lines and with simple structures. Therefore, recognizing those headers correctly is easier. Since the rule used to determine metadata lines is dependent only on the positions of headers, it is also easier to recognize metadata in these two datasets.

For *CRF*^{L} and *Strudel*^{L}, classifying header, group, derived and notes correctly is in general more challenging, compared to metadata and data, according to the per-class scores. Both algorithms perform better on *CIUS* than on the other datasets, because many files in this dataset are essentially the reports from different years on the same themes with the same templates – there are few file structure outliers. Both approaches do not work well on *derived* in *SAUS*, because the dataset has many unanchored derived cells. *GovUK* and *DeEx* are difficult to both approaches, because they both have many heterogeneous files regarding their structures.

In summary, *Strudel*^{L} outperforms *CRF*^{L} without using its stylistic features on our datasets, showing that our approach is more effective when fewer assumptions can be made about the input. *Strudel*^{L} is also more flexible than rule-based approaches such as *Pytheas*^{L} in predicting cases that are not covered by the given rule set.

6.2.2 Cell classification. For the cell classification task, we compare *Strudel*^{C} with two aforementioned algorithms: (i) *Line*^{C} provides a reasonable baseline, as most lines have homogeneous cells; (ii) *RNN*^{C} is based on an advanced deep learning architecture. The authors of *RNN*^{C} evaluated their approach also without stylistic features, which allows a fair comparison to *Strudel*^{C}. Table 6 (bottom) summarizes the comparative result in terms of the per-class and macro-average F1 score, and accuracy. *Strudel*^{C} surpasses its competitors . Meanwhile, the macro-average of *RNN*^{C} shows an advantage against the baseline approach, although the per-class scores of the two approaches are on par with each other. Even though cell classification is a more imbalanced task than line classification, the performance of our *Strudel*^{C} approach is comparable to its line counterpart.

Similar to the line classification problem, metadata, group, derived and notes are the difficult classes in general. *Group* cells are challenging for all approaches, as cells of this class are particularly rare. However, unlike other rare classes, such as metadata and notes, group cells are more likely to co-occur in the same line with data cells. *Line*^{C} reported low F1 score particularly on group and derived cells across datasets. In fact, both group and derived cells often co-occur with other types in the same lines: some tables contain a group cell in a line with several derived cells; other tables have derived columns rather than lines, therefore causing the few derived cells in a same line with multiple data cells. Group and derived cells usually account for a minor amount in these cases.

Line^C applies a majority-take-all strategy to extend the line prediction result of a line to all its non-empty cells, and therefore causes false negative for group and derived cells in the above two cases. *RNN*^{C} shows a low F1 score on the group class, which is not considered in the original paper [18], showing that the approach cannot be directly adapted to it. The set of the reforged derived cells, many of which were misplaced in the original annotations, is also troublesome for *RNN*^{C}, which does not involve value calculation mechanisms to detect them.

6.3 Strudel performance evaluation

In this section, we present experimental results to gain insights on the following questions: (i) When does Strudel mis-classify an instance of a particular class, and which class is most likely to be considered? (ii) Whether our approaches generalize to plain-text files that do not stem from spreadsheets? (iii) How do the features exploited in this work affect performance? (iv) What are the typical reasons that cause these incorrect predictions?

6.3.1 Line classification. Table 6 has introduced the per-class and overall F1 results of *Strudel*^{L}. In this section, we present our analysis of the classification results by using the confusion matrix. Figure 3 (top) shows the confusion matrix on executing *Strudel*^{L} per dataset. Due to space limitations, we leave out the matrix for *SAUS*, which is very similar to that for *GovUK*. To create a confusion matrix with the repeated 10-fold cross validation setting, we concatenate for each line in the files the predictions of all repetitions, and construct an ensemble prediction for it with the majority voting strategy. To resolve possible ties, we stipulate that the fewer instances of a class included in the dataset, the more prior the class is. We normalize the confusion matrix by the number of instances with particular classes.

Correctly identifying derived lines is the most challenging task across all datasets. These lines are mostly misclassified as data. The two main reasons for this are the lack of derived line training instances and the high similarity between derived and data lines, w.r.t. data types and spatial characteristics. Around 11.4% of the derived lines are treated as headers for *GovUK*. We observed this to happen in many tables where derived lines are between header and data areas and separated from these two areas by empty lines. Note that when a line of a minority (non-data) class is misclassified, the wrong prediction tends to be 'data', as the data class has much more instances than any other class. Apart from derived lines, header, group, and notes lines in *DeEx* also incorrectly lean towards the data class, because this dataset contains many tables of complicated structures. We discuss the kinds of mistakes in these categories in Section 6.3.6.

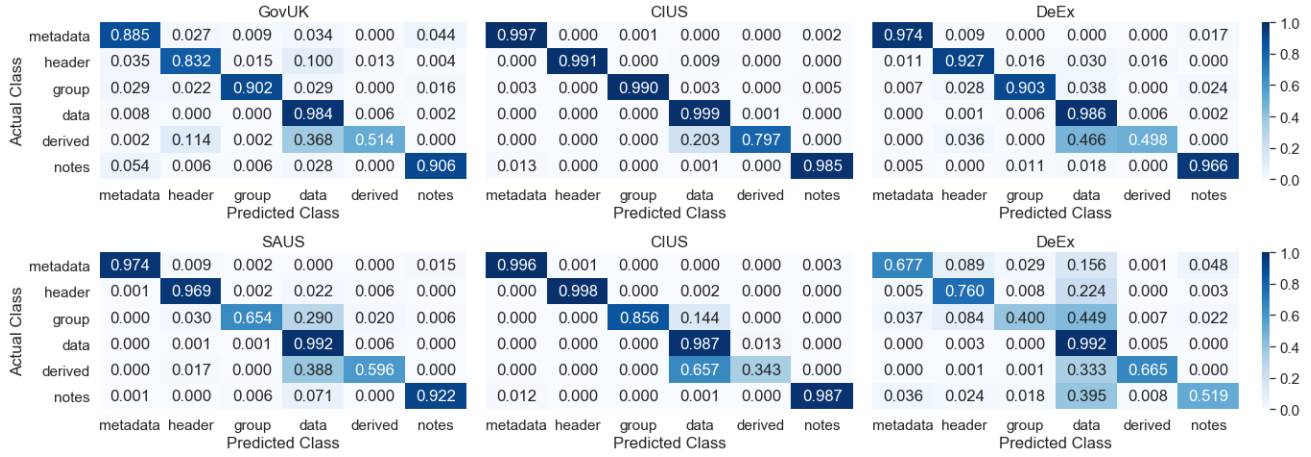


Figure 3: Confusion matrices to describe the pair-wise performance of *Strudel*^{L} (top) and *Strudel*^{C} (bottom) on individual datasets. The numbers are normalized by the amount of instances per class.

6.3.2 Cell classification. Similar to line classification experiments, we performed a set of experiments for evaluating *Strudel*^{C} and show the results in this section. Figure 3 (bottom) depicts the per-dataset confusion matrix on executing *Strudel*^{C}. To create it, we applied on the repeated cross validation results the same procedure used to create the confusion matrix for *Strudel*^{L}.

Compared to the confusion matrix of *Strudel*^{L}, more classes have a higher mis-classification ratio for all three datasets, showing that cell classification is a more challenging task than its line counterpart. On the one hand, the tendency to mark the instances of the minority classes as data is still prevalent. On the other hand, as the complexity of the problem increases, we do not observe many classification errors between two non-data classes, showing the effectiveness of our approach to distinguish between pairs of elements belonging to minority classes. About two-thirds of the derived cells are treated as data for *CIUS*. This is because a number of files share a fixed table schema that uses no keywords to indicate derived columns. Therefore, they are effectively ignored by the derived cell detection component.

6.3.3 Out-of-domain classification performance. To test the out-of-domain classification performance of Strudel, we kept the *Troy* dataset unseen during its design. We used a model trained on the collection of *SAUS*, *CIUS*, *DeEx* datasets to predict the line and cell classes of each file in this dataset.

The results in Table 7 show that group and derived cells are challenging for Strudel. After inspection, we found out that most of the derived cells lay in the lines that do not contain any derived keyword, such as 'total'. These derived cells are excluded from the candidate set, because our derived cell detection algorithm (Algorithm 2) relies on these keywords to anchor the candidates in order to reduce the search space. A typical derived line contains few group cells (usually the left-most) with indicative strings, such as 'total' and a number of numerical derived cells. Many of these derived lines are mis-classified as data, leading to the group cells therein also being mistaken.

6.3.4 Performance on plain-text files. To verify the effects of our Strudel algorithms on more difficult plain-text files that are not converted from spreadsheets, we tested the Strudel algorithm on the *Mendeley* dataset. We trained a model of our algorithm on the collection of *SAUS*, *CIUS*, and *DeEx* datasets, using the whole *Mendeley* dataset for testing.

Table 7: Per-class and overall F1 score on the *Troy* dataset.

	Strudel^L	*# lines*	*Strudel*^{C}	*# cells*
metadata	.935	317	.921	321
header	.798	278	.840	1,341
group	.667	42	.232	294
data	.937	2,898	.936	18,600
derived	.070	239	.216	1,935
notes	.971	575	.952	592
macro-avg	.730	4349	.683	23,083

Table 8 displays the per-class and overall F1 score for this experiment. As mentioned above, files of the *Mendeley* dataset are mostly used to store (tabular) data. Therefore, the minority classes in the *Mendeley* dataset have very few instances.

While the overall F1 scores in this experiment are inferior to the respective ones shown in Table 6, they do show that even for such difficult files our approaches are well able to distinguish data from non-data. The values in *Mendeley*'s plain-text files show properties different from traditional spreadsheets, e.g., length of metadata and notes areas, width of files. The second reason is that no data from *Mendeley* was included in the training phase. Therefore, dataset-specific properties are not learned properly by the classifiers. Last but not least, different areas in a plain-text file might have their own delimiters. As the delimiter of the table areas is used across the file, it is possible to destroy the intrinsic structures of other areas, e.g., when using comma as the delimiter, the value of a note line is split across multiple cells.

Regarding the results of individual classes, our model treats quite a few metadata lines as data, as the delimiter of metadata and data areas are different. At times, the delimiter dilemma also confuses our model of header lines in files where these lines are not split correctly. Out of the few derived cells, most are located in a single file, where the derived cells form a table by themselves, and aggregate on the values from another table, which is not recognizable by our model.

As the *Mendeley* dataset holds the biggest files across all our datasets, we also tested the scalability of our approach. The overall runtime on classifying cells of a file includes that of dialect

detection, feature creation, and cell class prediction. Our experiments show that the overall runtime is linear to the file size. For a file of around 10MB, the whole procedure takes around 256s on a 1.4 GHz MacBook Pro with 16GB RAM. Most of the time is spent on creating the feature vectors, which could be easily parallelized if possible. While we have few big files of such size, most files are only several kilobytes, probably because files with verbose information are usually used to show limited key information, rather than store a big amount of data.

Table 8: Per-class and overall F1 score for *Mendeley*.

	Strudel^L	# lines	*Strudel*^{C}	# cells
metadata	.623	604	.245	2,152
header	.406	86	.629	769
group	.263	27	.303	44
data	.999	194,786	.999	1,356,635
derived	.364	9	.051	99
notes	.448	86	.380	111
macro-avg	.517	195,598	.435	1,359810

6.3.5 Feature analysis. To understand which features exert more influence than others on particular classes, we calculated the feature importance for both *Strudel*^{L} and *Strudel*^{C} models. There are a variety of techniques to calculate feature importance [3, 21]. As many of our features are low-cardinality categorical features, we exploited permutation feature importance, because it does not favor high cardinality features [3]. Permutation feature importance indicates the ability of one feature to distinguish instances of one class from those of another in a binary classification scenario. To adapt this metric to our multiclass classification problem, we trained a model for each class in a one-vs.-rest fashion, and use the permutation feature importance of each such binary classifier to represent the ability of our model to detect instances of the particular class. The permutation of each feature was repeated five times and averaged.

Figure 4 illustrates the per-class feature importance for *Strudel*^{L} (top) and *Strudel*^{C} (bottom) with 100% stacked bars. The models are trained on the collection of *SAUS*, *CIUS*, and *DeEx*. We grouped all neighbor profile features (Section 5.3) into neighbor value length and neighbor data type to reduce the complexity of the figure, as each individual feature has little importance on the cell classification task. Up to five most important features whose proportions are higher than 10% are highlighted.

The line type probability feature is the most crucial feature for notes, metadata, and header. The percentage of empty cells in the row is also quite useful for notes and metadata. The percentage of empty cells in a column is most important to discover group cells: many group cells are in the left-most column (also indicated by the importance of the column position feature) of a file and span multiple rows. Neighbor profile features are most useful on discovering group cells, proving that group cells tend to locate in specific places. The novel feature signifying whether the value of a cell is the aggregation of other cells in the same line or column plays a great role in detecting derived cells, proving its effectiveness. Besides that, the existence of derived keywords, such as 'total' in the same column is also important, indicating that users tend to use these words to mark the derived columns. However, the existence of derived keywords in the same line shows quite limited importance in our experiment, although we expected similar importance of it as its column counterpart.

6.3.6 Analysis of difficult cases. The confusion matrices shed light on which classes are most commonly mis-predicted, either in the line or cell classification task. Here, we identify typical causes of those errors. The list below describes the pairs of common mis-classification cases (with > 10% incorrect classification in the class), e.g., mis-classifying 'derived as data', each followed by an error analysis after manually inspecting the results.

- **Derived as data.** Errors of this type usually happen because derived lines without keywords, such as 'total', in any of the cells are ignored by the derived cell detection algorithm, which uses these words to determine candidates, or because derived lines aggregate values from non-consecutive lines, which are ignored by the detection algorithm.
- **Header as data.** A header line with a number of non-textual values adjacent to a data line may be mis-classified as part of the data area. Examples include numeric headers, such as year and date. Files with multiple vertically-stacked tables may also be affected by this sort of error, as the headers of the tables towards the bottom of the stack have unusual line positions.
- **Notes as data.** Organizing notes as a small table is not uncommon, particularly in *DeEx*. Therefore, these tables of notes are likely to be treated as data. In some cases, authors place notes to the right of a table. Therefore, they are likely to be treated as data areas during cell classification.
- **Group as data.** One reason for this type of error is that some files have multi-level group columns, such as 'country-state-city', to the left of a table, followed by a number of data columns to the right. As most tables have few group columns, the classifier may mis-interpret these rare cases as data. Another reason is that these group cells lay in the same lines as those derived cells, who are not captured by the derived cell detection algorithm, because there is no keyword in the same row or column.
- **Metadata as data.** Tables may have elaborate metadata organized as small tables. Due to the tabular features of these metadata tables, Strudel tends to interpret them as data cells.

In summary, there are three aspects that mostly affect the correctness of our approach: (i) the geographical characteristic of vertically stacked multi-table files; (ii) the arithmetic calculation method for derived lines; (iii) the similarity between numeric header lines and data lines. These facts offer directions for improving our approach in the future work.

7 CONCLUSIONS

Often, valuable data are stored in semi-structured documents and cannot be directly parsed by common data management tools. Prior to extracting information from these files, it is necessary to understand their structure, by means of element classification, at either line or cell level. Previous works have addressed the line or cell classification problem for style-enriched documents, such as web tables or spreadsheets. In this work, we address both tasks on verbose CSV files that, similar to spreadsheets, organize data in a flexible layout, yet lack rich-text features. We addressed the two classification problems separately and designed a set of features for each of them, including content features, contextual features, and computational features.

Based on the experimental evidence, we discovered that with well-designed features, it is possible to reach decent performance of classifying lines and cells in a verbose CSV file and spreadsheets even if the stylistic features are not available. To conduct fair comparison between Strudel and related work, we use only the non-stylistic features. We summarize a handful of reasons that

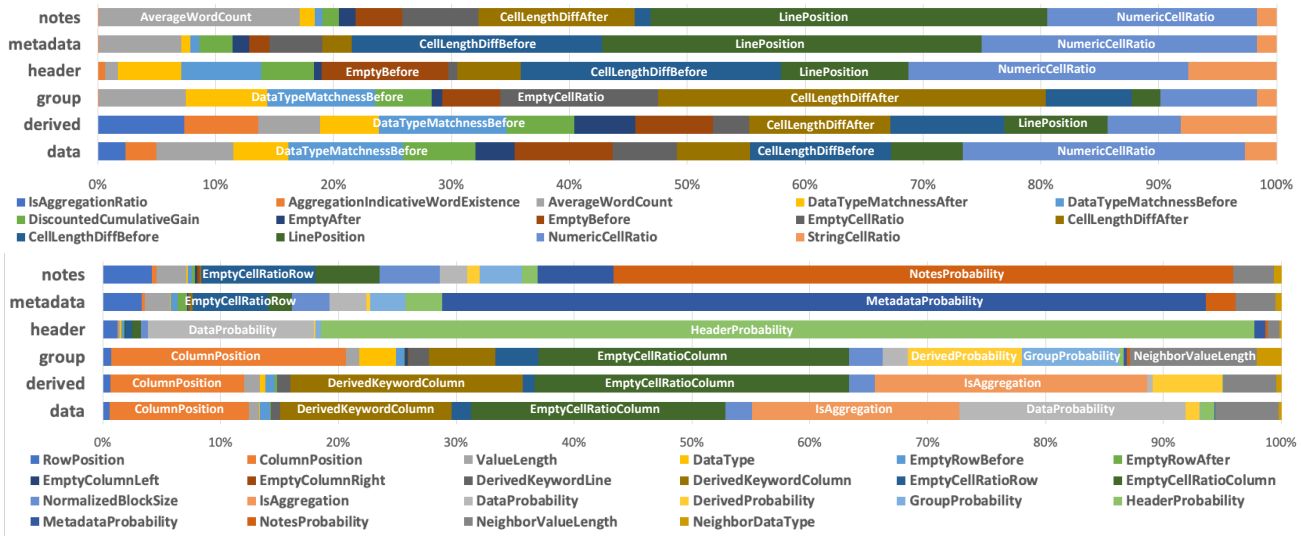


Figure 4: Feature importance of *Strudel*^{L} (top) and *Strudel*^{C} (bottom) trained on the collection of *SAUS*, *CIUS*, and *DeEx*. Several most important features for each class are highlighted.

cause common misclassification cases, and recognize the effectiveness of computational features that are neglected by former studies, drawing key insights for further structure understanding research: (i) how to improve the prediction quality with semantic features; (ii) how can we extend the derived cell detection algorithm by recognizing more aggregation functions; (iii) whether column classification can help boost the classification quality.

REFERENCES

[1] Robin Abraham and Martin Erwig. 2007. UCheck: A spreadsheet type checker for end users. *Journal of Visual Languages & Computing* 18, 1 (2007), 71–95.

[2] Marco D Adelfio and Hanan Samet. 2013. Schema extraction for tabular data on the web. *PVLDB* 6, 6 (2013), 421–432.

[3] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[4] Michael J Cafarella, Alon Y Halevy, Yang Zhang, Daisy Zhe Wang, and Eugene Wu. 2008. Uncovering the Relational Web.. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*.

[5] Zhe Chen and Michael Cafarella. 2013. Automatic web spreadsheet data extraction. In *International Workshop on Semantic Search over the Web*. 1–8.

[6] Zhe Chen, Mike Cafarella, Jun Chen, Daniel Prevo, and Junfeng Zhuang. 2013. Senbazuru: A Prototype Spreadsheet Database Management System. *PVLDB* 6, 12 (2013), 1202–1205.

[7] Zhe Chen, Sasha Dadiomov, Richard Wesley, Gang Xiao, Daniel Cory, Michael Cafarella, and Jock Mackinlay. 2017. Spreadsheet property detection with rule-assisted active learning. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 999–1008.

[8] Christina Christodoulakis, Eric Munson, Moshe Gabel, Angela Demke Brown, and Renée J. Miller. 2020. Pytheas: Pattern-based Table Discovery in CSV Files. *PVLDB* 13, 11 (2020), 2075–2089.

[9] Xu Chu, Yeye He, Kaushik Chakrabarti, and Kris Ganjam. 2015. Tegra: Table extraction by global record alignment. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1713–1728.

[10] Cristian Consonni, Paolo Sottovia, Alberto Montresor, and Yannis Velegrakis. 2019. Discovering order dependencies through order compatibility. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 409–420.

[11] Haoyu Dong, Shijie Liu, Shi Han, Zhouyu Fu, and Dongmei Zhang. 2019. TableSense: Spreadsheet Table Detection with Convolutional Neural Networks. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 69–76.

[12] Haoyu Dong, Shijie Liu, Shi Han, Zhouyu Fu, and Dongmei Zhang. 2019. Tablesense: Spreadsheet table detection with convolutional neural networks. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Vol. 33. 69–76.

[13] Julian Eberius, Christoper Werner, Maik Thiele, Katrin Braunschweig, Lars Dannecker, and Wolfgang Lehner. 2013. DeExcelerator: a framework for extracting relational data from partially structured documents. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 2477–2480.

[14] Hazem Elmeleegy, Jayant Madhavan, and Alon Halevy. 2009. Harvesting relational tables from lists on the web. *PVLDB* 2, 1 (2009), 1078–1089.

[15] David W Embley, Mukkai S Krishnamoorthy, George Nagy, and Sharad Seth. 2016. Converting heterogeneous statistical tables on the web to searchable databases. *International Journal on Document Analysis and Recognition* 19, 2 (2016), 119–138.

[16] Wolfgang Gatterbauer, Paul Bohunsky, Marcus Herzog, Bernhard Krüpl, and Bernhard Pollak. 2007. Towards domain-independent information extraction from web tables. In *Proceedings of the International World Wide Web Conference (WWW)*. 71–80.

[17] George Nagy. 2016. TANGO-DocLab web tables from international statistical sites (Troy_200). http://tc11.cvc.uab.es/datasets/Troy_200_1.

[18] Majid Ghasemi-Gol, Jay Pujara, and Pedro Szekely. 2019. Tabular Cell Classification Using Pre-Trained Cell Embeddings. *Proceedings of the International Conference on Data Mining (ICDM)* (2019).

[19] Elvis Koci, Maik Thiele, Óscar Romero Moral, and Wolfgang Lehner. 2016. A machine learning approach for layout inference in spreadsheets. In *IC3K: Proceedings of the International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*. SciTePress, 77–88.

[20] Ying Liu, Prasenjit Mitra, and C Lee Giles. 2008. Identifying table boundaries in digital documents via sparse line detection. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 1311–1320.

[21] Wei-Yin Loh. 2011. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 1, 1 (2011), 14–23.

[22] Fatemeh Nargesian, Erkang Zhu, Ken Q Pu, and Renée J Miller. 2018. Table union search on open data. *PVLDB* 11, 7 (2018), 813–825.

[23] David Pinto, Andrew McCallum, Xing Wei, and W Bruce Croft. 2003. Table extraction using conditional random fields. In *Proceedings of the International Conference on Information retrieval (SIGIR)*. 235–242.

[24] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. 2019. DynFD: Functional Dependency Discovery in Dynamic Datasets.. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 253–264.

[25] Alexey O Shigarov and Andrey A Mikhailov. 2017. Rule-based spreadsheet data transformation from arbitrary to relational tables. *Information Systems (IS)* 71 (2017), 123–136.

[26] Saravanan Thirumuruganathan, Nan Tang, Mourad Ouzzani, and AnHai Doan. 2020. Data Curation with Deep Learning.. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 277–286.

[27] Gerrit JJ van den Burg, Alfredo Nazábal, and Charles Sutton. 2019. Wrangling messy CSV files by detecting row and type patterns. *Data Mining and Knowledge Discovery* 33, 6 (2019), 1799–1820.

[28] Alexander Wachtel, Michael T Franzen, and Walter F Tichy. 2016. Context Detection in Spreadsheets Based on Automatically Inferred Table Schema. In *International Conference on Human-Computer Interaction*.

[29] Yalin Wang and Jianying Hu. 2002. A machine learning based approach for table detection on the web. In *Proceedings of the International World Wide Web Conference (WWW)*. 242–250.

[30] Meihui Zhang and Kaushik Chakrabarti. 2013. InfoGather+: semantic matching and annotation of numeric and time-varying attributes in web tables. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 145–156.

FRESQUE: A Scalable Ingestion Framework for Secure Range Query Processing on Clouds

Hoang Van Tran[1,3], Tristan Allard[1], Laurent d'Orazio[1], Amr El Abbadi[2]
[1]Univ Rennes & CNRS & IRISA, [2]UC Santa Barbara, [3]Can Tho University

ABSTRACT

Performing non-aggregate range queries over encrypted data stored on untrusted clouds has been considered by a large body of work over the last years. However, prior schemes mainly concentrate on improving query performance while the scalability dimension still remains challenging. Due to heavily pre-processing incoming data at a trusted component such as encrypting data and building secure indexes, existing solutions cannot provide a satisfactory ingestion throughput. In this paper, we overcome this limitation by introducing a framework for secure range query processing, FRESQUE, that enables a scalable consumption throughput while still maintaining strong privacy protection for outsourced data. Our experiments on real-world datasets show that FRESQUE can support over 160 thousand record insertions in a second, when running on a 12-computing node cluster. It also significantly outperforms one of the most efficient schemes such as PINED-RQ++ by 43 times on ingestion throughput.

1 INTRODUCTION

With the prosperity of online social networks and web-based services, a large amount of personal data is collected every second. To achieve analytical and administrative purposes, it becomes increasingly desirable for modern systems to support not only low-latency queries, but also intensive ingestion throughput over incoming data. For instance, to reduce the impact of seasonal epidemics (e.g., influenza), participatory surveillance systems, to name few, have been deployed in recent years such as Flu Near You [2] in North America, Influenzanet [16] in Europe, and FluTracking [3, 9] in Australia and New Zealand. These systems weekly collect symptoms from their participants to track influenza nation-wide. Such systems are expected to receive a huge number of records every second.

Due to the significant costs necessary for building and maintaining such systems (e.g., computing and scalability requirements, human resources), it may be worth to outsource user data to a cloud service, that can provide lower costs and enable elastic scaling [14]. However, parts of the data may be sensitive, e.g., participants' symptoms, and sociodemographic data (age, gender, etc.). Managing sensitive data on a public cloud increases the risk of unauthorized disclosure since its infrastructure may be compromised by an adversary. According to a recent survey, 52% of companies use cloud services that have experienced a data breach [18].

This paper considers the following cloud computing model (Figure 1): a collector receives data from multiple data sources and stores them on a cloud while a client retrieves the data by using queries. For a simple example, Flu survey participants submit electronic medical records to a Center for Disease Control and Prevention (CDC) that stores the collected data on a cloud. An epidemiologist queries these records from the cloud. In this model, only the cloud is untrusted while the others are trusted.

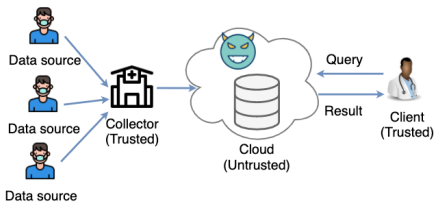


Figure 1: An architecture of cloud computing

Encryption is a standard technique to ensure the confidentiality of data stored on untrusted environments like clouds [29, 32, 35]. In this study, we focus on non-aggregate range queries over encrypted data since they are fundamental operations. For example, a doctor, Alice, wants to get encrypted electric records of the patients whose ages are between a and b. One example of a range query (expressed with SQL) can be: SELECT * FROM database WHERE age $\geq$ a AND age $\leq$ b.

Although many studies in this line of work have been done over the last years [5–8, 10, 19, 20, 24, 26, 30, 31, 36], none of them satisfies real-life scalability requirements. It is also non-trivial to scale these schemes due to their own limitations. For instance, Hidden Vector Encryption (HVE) methods [8, 36] use bilinear groups equipped with bilinear maps and hide attributes in an encrypted vector. However, they suffer from high latency because it is extremely costly to compute exponentiation and pairing in a composite-order group. Meanwhile, some recent schemes [10, 23, 24] attempt to maintain secure indexes, that rely on *Searchable Symmetric Encryption*, for efficiency. Unfortunately, the secure indexes not only create high space overhead, but also require at least hundreds of second for construction that may lead to bottlenecks. Even though ArxRange [30] does not experience long index building time, it incurs prohibitive storage overhead and only supports a modest ingestion throughput, e.g., about 450 writes per second with caching. Recently, solutions based on differential privacy [11], e.g., the PINED-RQ family [33, 34], have been considered and achieve very good performance in terms of computation and space requirements, however, they do not render a satisfactory ingestion throughput. In particular, PINED-RQ [33] incurs congestion as incoming data rate is high. Meanwhile, PINED-RQ++ [34] experiences modest throughput, ~46K records/s.

Therefore, we aim at developing a scalable ingestion framework dedicated to secure range query processing. To the best of our knowledge, this is the first paper about the architecture of such system. Our solution is developed based on the PINED-RQ family [33, 34]. This choice is motivated by the fact that this family can achieve both fast range queries and provable security guarantees while the secure index requires small space. More specifically, we re-design the architecture of PINED-RQ++ [34] in order to make it fully distributed. That is, we attempt to distributively process all heavy tasks (e.g., parsing and encrypting data) on a

set of *shared-nothing* machines[1]. By relying on such architecture, the system can easily scale. Additionally, we introduce a data representation and an asynchronous publishing method to decrease throughput degradation as much as possible. By precisely coordinating all of them together, our framework can enable intensive ingestion throughput. Experimental results show that as compared to (non-)parallel PINED-RQ++ [34], the throughput is improved by (~43×) ~5.6× respectively (NASA dataset [1]). Furthermore, we also present a new noise management mechanism to cope with strong online attackers having background knowledge about the time distribution of incoming data. This method also improves the practicality of FRESQUE since it does not require a pre-defined timestamp distribution as in PINED-RQ++ (see Section 4.1). In this study, we develop **FRESQUE**, a scalable ingestion **fr**am**e**work for **s**ecure range **que**ry processing on cloud, including the following contributions.

(1) We thoroughly analyze the architecture of the PINED-RQ family [33, 34], and point out obstacles that prevent the existing architecture from achieving a high ingestion throughput. Also, we propose an approach to cope with offline and online attackers while minimizing the required knowledge of the collector.
(2) We design an ingestion architecture that enables to distribute all heavy jobs to multiple workers (computing nodes) of a cluster. Besides, we present and integrate a data representation and an asynchronous publishing method to this architecture, mitigating throughput degradation.
(3) We extensively evaluate FRESQUE on real-world datasets to demonstrate its scalability. Particularly, the throughput of FRESQUE is about 43× higher than that of PINED-RQ++ and being at least one order of magnitude higher than that of other efficient solutions such as [6, 30, 31].
(4) We formally analyze the security of FRESQUE.

The paper is structured as follows. In Section 2, we briefly introduce the problem statement. Section 3 reviews the related work. We analyze the architecture of the PINED-RQ family in Section 4. We then describe our framework in Section 5. Section 6 gives security analyses while Section 7 presents our experimental results. We discuss an application of our solution in Section 8. Section 9 concludes the paper and gives future work.

2 PROBLEM STATEMENT

We assume that data sources produce a set of records where all records have the same number of attributes. These records are immediately sent to the collector. The dataset stored at the collector is a relation $D(A_1, \ldots, A_n)$, where A_i is an attribute. Queries are non-aggregate one-dimensional range queries. A query Q is evaluated over the attribute A_q of D, which contains numerical values. Periodically, the collector pre-processes the dataset, e.g., building a secure index over the dataset and encrypting it, prior to sending it to the cloud.
In this study, we concentrate on the scalability dimension of the system in terms of ingestion throughput. This metric measures how many records a system is able to consume within a time period. The target solution should meet additional requirements, namely formal security guarantees, supporting updates, and incurring practical storage overhead.

[1]We take the shared-nothing architecture into account since it is highly scalable.

2.1 Threat model

We consider the *honest-but-curious* model [15]. In this model, an attacker examines data stored on the cloud to glean sensitive information, but follows the protocol as specified and does not change the datasets or query results. Also, we consider three types of attackers : (1) the *offline* attacker is able to access a copy of the encrypted datasets and secure indices (e.g., by stealing the hard drives), (2) the *online* attacker is able to observe any information available at the cloud or being exchanged between the cloud and the trusted components, and (3) the *informed online* attacker is an online attacker that further has background knowledge about the data distribution of the incoming time of real data.

2.2 Security

2.2.1 Differential privacy. The ϵ-differential privacy model [11] requires that any possible individual record can only have a limited impact on the output distribution of an ϵ-differentially private function. This model considers a very strong adversary that is not computationally-bounded (information-theoretic guarantees). Definition 1 gives a formal definition.
Definition 1 (ϵ-differential privacy [11, 13]): A randomized mechanism $\mathcal{M}$ satisfies ϵ-differential privacy, if for any set $O \in Range(\mathcal{M})$, and any datasets $\mathcal{D}$ and $\mathcal{D}'$ s.t. $\mathcal{D}$ is $\mathcal{D}'$ with one record more or one record less,

$$Pr[\mathcal{M}(\mathcal{D}) = O] \leq e^{\epsilon} \cdot Pr[\mathcal{M}(\mathcal{D}') = O]$$

where ϵ represents the privacy level. A smaller ϵ means stronger privacy level. The Laplace mechanism is the most common method to achieve ϵ-differential privacy.
Laplace Mechanism [12]: Let $\mathcal{D}$ and $\mathcal{D}'$ be two datasets such that $\mathcal{D}$ is $\mathcal{D}'$ with one record more or one record less. Let $Lap(\beta)$ be a random variable that has a Laplace distribution with the probability density function $pdf(x, \beta) = \frac{1}{2\beta} e^{-|x|/\beta}$. Let f be a real-valued function, the Laplace mechanism adds $Lap(max \parallel f(\mathcal{D}) - f(\mathcal{D}') \parallel_1 / \epsilon)$ to the output of f, where $\epsilon > 0$.
Theorem 1 (Sequential Composition [27]): Let $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_r$ denote a set of mechanisms and each $\mathcal{M}_i$ gives ϵ_i-differential privacy. Let $\mathcal{M}$ be another mechanism that executes the sequence of $\mathcal{M}_1(\mathcal{D}), \mathcal{M}_2(\mathcal{D}), \ldots, \mathcal{M}_r(\mathcal{D})$. Then $\mathcal{M}$ satisfies $(\sum_{i=1}^{r} \epsilon_i)$-differential privacy.

2.2.2 Semantic Security. Loosely speaking, a cryptosystem is semantically secure if it is infeasible for a computationally-bounded adversary, i.e., a probabilistic polynomial algorithm, to derive significant information about plaintext from its ciphertext and any auxiliary information, e.g., obtained from external sources. Today, AES (in CBC mode) is the common instance of efficient private key encryption schemes satisfying semantic security.
Definition 2 (Semantic security [15]): A private key encryption algorithm E_χ, where χ is the secret key, is semantically secure if for every probabilistic polynomial time algorithm A there exists a probabilistic polynomial time algorithm A' such that for every input dataset $\mathcal{D}$, every auxiliary background knowledge $\zeta \in \{0,1\}^*$, every polynomially bounded function $g : \{0,1\}^* \rightarrow \{0,1\}^*$, every polynomial $p(\cdot)$, every sufficiently large $n \in N$, it holds that:

$$Pr[A_n(E_\chi(\mathcal{D}), |\mathcal{D}|, \zeta) = g(\mathcal{D})] < Pr[A'_n(|\mathcal{D}|, \zeta) = g(\mathcal{D})] + \frac{1}{p(n)}$$

2.2.3 Unified privacy model. Sahin *et al.* [33] is based on a probabilistic relaxation of a variant of differential privacy that considers computationally bounded adversaries [28] and that considers the cryptographically-negligible leaks due to the use of efficient real-world encryption schemes, i.e., AES in CBC mode. Definition 3 is a simplification of ϵ_n-SIM-CDP, the simulation-based computational differential privacy model proposed in [28].

Definition 3 (ϵ_n-SIM-CDP privacy [28]): The randomized function f_n provides ϵ_n-SIM-CDP if there exists a function F_n that satisfies ϵ_n-differential-privacy and a polynomial $p(\cdot)$, such that for every input dataset $\mathcal{D}$, every probabilistic polynomial time adversary A, every auxiliary background knowledge $\zeta \in \{0,1\}^*$, and every sufficiently large $n \in N$, it holds that:

$$Pr[A_n(f_n(\mathcal{D},\zeta)) = 1] - Pr[A_n(F_n(\mathcal{D},\zeta)) = 1] \leq \frac{1}{p(n)}$$

Definition 4 ((ϵ,δ)-Probabilistic-SIM-CDP [33]): A randomized function f_n satisfies (ϵ,δ)-Probabilistic-SIM-CDP, if it provides ϵ_n-SIM-CDP to each individual with probability greater than or equal to δ, where $\delta \in [0,1]$.

3 RELATED WORK

In this section, we briefly review prior schemes with respect to the target requirements. Table 1 gives the corresponding summary.

Table 1: Prior schemes with respect to target requirements

Scheme	Formal security guarantees	Update support	Low latency	Small storage overhead
HVE [8, 36]	✓	✓		
Bucketization [17, 19, 20]		✓	✓	✓
OPE [5–7, 26, 31]		✓	✓	✓
PBtree [24]	✓		✓	
IBtree [23]	✓	✓	✓	
ArxRange [30]		✓	✓	
Demertzis et al. [10]	✓	✓	✓	
PINED-RQ family [33, 34]	✓	✓	✓	✓

Hidden vector encryption (HVE) methods [8, 36] employ asymmetric cryptography to conceal attributes in an encrypted vector. A range predicate is privately evaluated on such vector. However, these schemes incur prohibitive computation and storage costs.

Bucketization approaches [17, 19, 20] partition an attribute domain into a finite number of buckets. Each bucket is then assigned by a random tag (bucket-id). When the client sends a range query to the server, the buckets that intersect the query are determined by using the index tag stored at the client. All contents of the intersecting buckets are finally returned to the client. These approaches lack formal privacy guarantee.

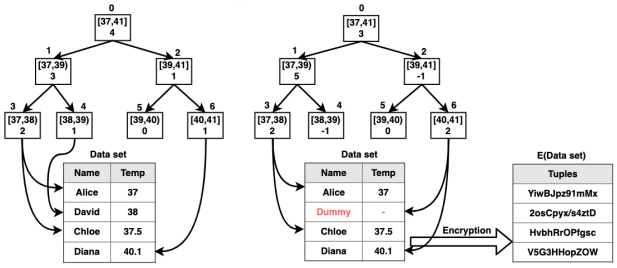
Order-preserving encryption schemes (OPE) [5–7, 26, 31] transform plaintexts into ciphertexts so that the relative order of their plaintexts is preserved. This property enables to efficiently execute range predicate evaluation on encrypted data. Unfortunately, OPE schemes disclose the underlying data distribution, and hence they are vulnerable to statistical attacks.

Recently, a few works have taken indexing solutions into account such as [10, 23, 24, 30, 33]. These schemes seek to maintain a secure index over outsourced data for fast range query processing. However, most of them [10, 23, 24, 30] suffer from prohibitive storage overhead. On the other hand, Sahin et al. [33] introduce an approach constructing *clear* secure indexes, called PINED-RQ indexes, for serving efficient range queries. The efficiency is enabled by the secure index while the security relies on computational differential privacy guarantees. Although PINED-RQ index is small in space, this scheme publishes data in batches, hence suffering from bottlenecks in the system as data arrives at high speed. Tran et al. [34] later introduce an index template-based approach, PINED-RQ++, to handle this limitation. Similar to prior schemes, high ingestion throughput still remains challenging for these solutions.

As shown in Table 1, the PINED-RQ family outperforms its counterparts with regard to the target requirements. To achieve our goal, we thus develop our solution based on this family. Particularly, we address the drawback of modest throughput in PINED-RQ.

4 ANALYSIS OF THE PINED-RQ FAMILY

This section gives thorough analyses of the PINED-RQ family [33, 34] and points out their drawbacks. Based on them, we develop our solutions in the Section 5.



(a) Clear index **(b) Perturbed index**

Figure 2: Sample PINED-RQ index

4.1 Overview

Building index. There are two primary steps to build a secure PINED-RQ index.

Step 1 - Building an index. PINED-RQ is inspired from B+Trees. In PINED-RQ, the set of all nodes is defined as a histogram covering the domain of an indexed attribute (e.g., the participants' body temperature (Temp)), as illustrated in Figure 2a. Each leaf node has a count that represents the number of records falling within its interval. It also keeps pointers to those records. Likewise, the root and any internal node have a range and a count, combining the intervals and the counts of their children, respectively.

Step 2 - Perturbing an index. All counts in the index are independently perturbed by Laplace noise [12]. The noise may be positive or negative, thereby after this step, the count of a node may increase or decrease, respectively. As shown in Figure 2b, the count of node 4 changes from 1 to -1 while the count of node 6 changes from 1 to 2. Such changes consequently lead to inconsistencies between the noisy count of a leaf node and the number of pointers it holds. To address this issue, PINED-RQ adds dummy records to the dataset when a leaf node receives positive noise, otherwise, if a leaf node receives negative noise, real records are removed from the dataset. The records removed are then inserted into the corresponding *overflow array* which is a fixed-size array. This overflow array is filled with dummy records if it has free space. As illustrated in Figure 2b, the record (David) belonging to node 4 is removed from the dataset while one dummy record is added and linked to node 6. Lastly, the perturbed index, the encrypted dataset, and the overflow arrays are published to the cloud.

Query processing on indexed data. A range query will start from the root of an index. It then traverses the child of any node that has a non-negative count and intersects with the query range. This is repeated recursively until the leaves of the index are reached. At the leaves that overlap the query range, their records and overflow arrays are returned.

Building index with index template. To adapt PINED-RQ to

the context of high rate of incoming data, Tran et al. [34] have previously developed PINED-RQ++ that builds a secure index based on the notion of *index template*. In particular, the collector initially creates an index template and perturbs it by using Laplace noise. This means that the count of bins at first contains only noise. The real count of bins will be updated during a *publishing time interval*, which is defined as the period from when an index template is created to when it is published. During a publishing time interval, whenever a new record arrives, the index template is updated with the record. Next, the collector encrypts and forwards this record to the cloud. At the end of each publishing time interval, the updated index template is published, and the cloud associates it with earlier published records to produce a secure index.

To manage positive noise, being represented by dummy records, the collector can send dummy records to the cloud according to the actual distribution of the sending time of the real records. On the other hand, for negative noise, if a leaf node initially receives negative noise c, the collector moves the first c records (when they arrive) of that leaf node to the corresponding overflow array.

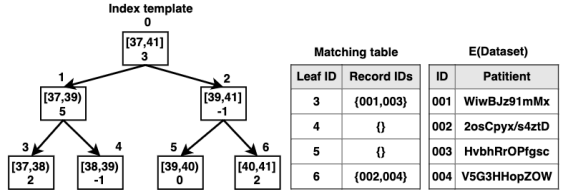


Figure 3: Perturbed index template and matching table (the corresponding PINED-RQ index is presented in Figure 2b)

To privately keep the relationships between published records and index template leaves, PINED-RQ++ utilizes a *matching table* (see Figure 3). In particular, a record is tagged by a random number instead of the real id of a leaf prior to being sent to the cloud. When the matching table is published at the end of each publishing time interval, the cloud uses it to reconstruct pointers between leaves and published data.

Generally, the workflow at the collector starts with initiating a new index template. When new data arrives, it sequentially passes a series of components, as depicted in Figure 4.

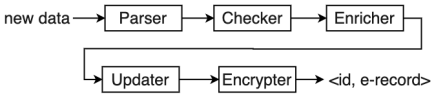


Figure 4: Workflow at the collector of PINED-RQ++

- **Parser** transforms incoming data into a pre-defined format.
- **Checker** buffers the parsed record at the collector as its indexed attribute belongs to a negative leaf. The index template is then updated. Otherwise, that record is forwarded to the next component.
- **Enricher** adds a random number (id) to the record.
- **Updater** updates the index template and matching table based on the record.
- **Encrypter** encrypts the record and gets e-record (encrypted record). The encrypter finally sends a pair of <id, e-record> to the cloud.

Parallel architecture of PINED-RQ++. Since the heavy workflow greatly degrades the throughput at the collector, Tran et al. [34] have introduced a parallel version of PINED-RQ++. Its overview architecture is depicted in Figure 5. Parallel PINED-RQ++ distributively processes heavy tasks on a set of independent

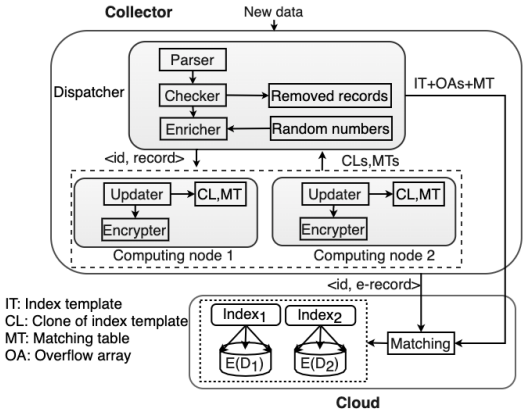


Figure 5: Architecture of parallel PINED-RQ++

machines (e.g., computing nodes), namely updater and encrypter. As a result, ingestion throughput exhibits a significant improvement. Nonetheless, several challenges still remain in this architecture. We now identify the obstacles that hinder PINED-RQ++ from achieving a scalable solution.

4.2 Limitations

Partial parallelism. In PINED-RQ++, the index template is proposed to temporarily store information that is necessary to build the secure index later. By doing it, the count variables of the index template are updated and referenced by the updater and the checker, respectively, during a publishing time interval. The index template is thus considered as a *shared* data structure, that does not run simultaneously in parallel PINED-RQ++. Furthermore, the checker depends on the parser for the checking operation. Hence, both parser and checker are organized to run in sequence at the collector (see Figure 5). Unfortunately, the parsing task usually takes time, especially in case of large record size. Thus, the parser mainly makes the ingestion throughput of the parallel PINED-RQ++ incredibly degraded. For instance, our experiments shows that the parsing task reduces the throughput of the collector by over 50% when we use NASA dataset [1].

Heavily updating index template. Since PINED-RQ++ uses the whole index template for updating and checking incoming data, there are some unnecessary overheads at the collector in terms of memory usage and computation. For example, an update always requires traversing from the root to leaves of the index template, having a complexity of $O(\log_k n)$, where n is the number of leaves and k is the branching factor. Likely, the checker faces the same complexity for checking a record whether it belongs to negative leaf or not. These tasks will take time to process records and diminish the ingestion ability of the system, especially when the domain of index template is huge. The situation is even worse when all tasks which reference the index template have to be processed sequentially.

Synchronous publication. Both PINED-RQ++ and its parallel version are designed to *synchronously* publish datasets to the cloud. In other words, they will start a new publication only if the current publication is sent to the cloud. This mechanism may create congestion in some circumstances. For instance, at the end of each publishing time interval, the collector needs to generate overflow arrays whose size primarily relies upon several configurable parameters, namely domain size, security level (e.g., ϵ), and bin interval. These parameters vary for different applications, thereby the size of overflow arrays will also change accordingly. As the size of overflow arrays is large, the collector

spends long time for generating overflow arrays, giving a heavy burden on the ingestion performance or even bottlenecks at this component.

5 INGESTION FRAMEWORK FOR SECURE RANGE QUERY (FRESQUE)

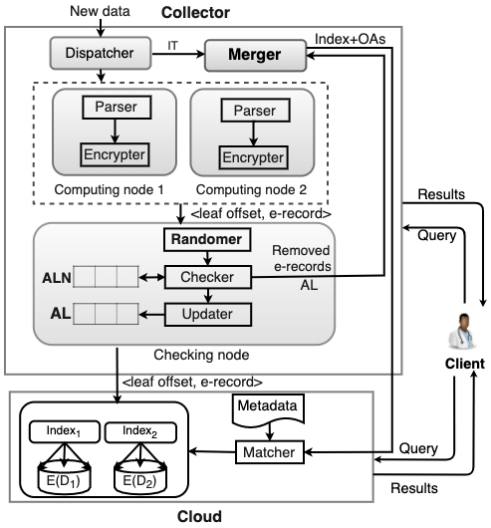


Figure 6: Architecture of FRESQUE

We first present below the key design features of FRESQUE that tackle the main limitations of PINED-RQ++ [34] (see Section 4.2). Then we describe how FRESQUE copes with the informed online attacker (see Section 2.1). Finally we present the complete architecture of FRESQUE.

5.1 Key Design Features

(a) A fully parallel architecture. As stated earlier, **partial parallelism** mainly causes low ingestion throughput in PINED-RQ++. To deal with that, we aim at making the collector fully distributed by parallelizing all heavy jobs (e.g., parser and encrypter) on a cluster of computing nodes.

The difficulty is that the checker, that resides between the parser and the encrypter in the workflow, cannot be parallelized since it relies not only on the parser but also on a shared data structure (e.g., index template). This means that the checker should be positioned after the parser and cannot be run in parallel. In fact, we can run the parser and the encrypter on multiple computing nodes while the checker runs sequentially at another node[2]. After incoming records are parsed at the instances of the parser, they are sent to the checker. These records are then checked by the checker before being sent back to the instances of the encrypter. Nevertheless, this approach would increase unnecessary communication overheads among components at the collector. Instead, we position the checker after the parser and the encrypter in the workflow, as illustrated in Figure 6. This approach allows to scale the intake ability of the collector without creating unnecessary transmission overheads. We then add additional information (e.g., leaf offset) to the ciphertext of the record so that the checker can know which leaf the record belongs to.

(b) Array representation of Leaves (AL). To address the problem of **heavily updating index template**, we replace the index template by an *array representation* of its leaves for the updating and checking operations in our new architecture. Such array representation is small to keeps in memory and accessing array elements requires constant time, $O(1)$. Particularly, the collector maintains two arrays of integers, one with noise (ALN) and the other without noise (AL). The former is used to check whether a record falls within a negative leaf or not while the latter is mainly used to count the number of real records passing the collector. Each element of AL/ALN represents the true count/noise of a leaf, respectively. The size of the two arrays is equal to the number of leaves of the index template. Note that the AL contains the true count of leaves while the IT only contains noise, thereby the two components are sufficient to compute the secure index.

To integrate such data representation into the new architecture, for a given value, the collector needs to know the *leaf offset* of the corresponding element in AL(N). Thanks to the strongly constrained shape of the PINED-RQ index, the leaf offset of a record can be easily obtained based on the configurable parameters of the system. Given parameters, namely domain min (d_{min}), domain max (d_{max}), bin interval (I_b), and an indexed attribute value (v), the leaf offset (O_v) of v can be achieved as follows.

$$O_v \leftarrow min(\lfloor (v - d_{min})/I_b \rfloor, \lfloor (d_{max} - d_{min})/I_b \rfloor - 1)$$

With such an approach, the checker is lightweight enough to avoid performance bottlenecks even if it runs sequentially.

(c) Asynchronously publishing mechanism. To address the issues of the **synchronous publishing** mechanism, we design our new architecture to *asynchronously* publish datasets. To this purpose, we add a new component to our architecture, named merger, that runs independently of the ingestion component (e.g., dispatcher), as depicted in Figure 6. The merger is only responsible for *publishing tasks*, namely building overflow arrays and combining a secure index from the AL and the IT (Index Templates). At the end of each publishing time interval, the publishing tasks are shifted to the merger, and a new publication is immediately initiated at the dispatcher. With this approach, while the dispatcher ingests data for a new publication, the merger performs the publishing tasks for the previous one. This eliminates the burden of the publishing tasks on the ingesting component and prevents potential bottlenecks at the collector. More importantly, the asynchronous publishing method allows the system to continuously consume incoming data with a very small latency for starting a new publication. Such property partially improves the ingestion throughput.

By using the asynchronous publication strategy, all components in FRESQUE, including the dispatcher and the merger, run independently. To ensure data consistency among publications, e.g., how a component determines to which publication a record belongs, we mark each publication with a unique *monotonic* number, named publication number.

5.2 Upgrading PINED-RQ++ for Coping with Informed Online Attackers

Information about the noise injected, e.g. dummy/removed records of a publication, may be disclosed to the informed online attacker if the order of the incoming data at the cloud is the same as the order of the incoming data at the collector. Indeed, since the informed online attacker has the time at which records (dummy or true) arrive at the cloud, his background knowledge on the time distribution of real data can enable him to distinguish (probabilistically) between incoming true records and incoming dummy records (positive noise), or to gain partial knowledge of the number of removed records (negative noise). First, the records removed by the checker at the collector (if they fall in the interval of a negative leaf - see Section 4) do not leave the collector before

[2]The encrypter can further benefit from hardware cryptographic modules.

the end of the publishing time interval. Their absence may reveal to the informed online attacker information about the values of negative noises. Second, the arrival of records at the cloud at unexpected times makes them more likely to be dummy record (information about positive noises). In PINED-RQ++, the collector releases dummy records according to the true distribution of the incoming time of real data for confusing the informed online attacker. This obviously requires to know the distribution in advance, which may be difficult to achieve in real-life applications. We now seek to design a new noise management method that mitigates privacy leaks against the informed online attacker and does not depend on any pre-defined distribution of the incoming time of real data.

To address this issue, we introduce a new component, called *randomer*, to our architecture (see Figure 6). It aims at *perturbing* the distribution of the incoming time of real data at the collector so that the insertion of dummy records or the deletion of real records are hidden from the adversary. The randomer consists of a *fixed-size buffer* and a *trigger* function. The former is used to *mix* incoming real and dummy records together while the latter is used to control the size of the buffer. In particular, all dummy records of a publication are first generated and are *uniformly at random* sent to the buffer of the randomer during a publishing time interval. For example, suppose a publication has 100 dummy records, then 100 time points are chosen uniformly at random over the current publishing time interval, and one dummy record is released at each time point. When a record (real/dummy) arrives at the randomer, it is buffered here. If the randomer buffer is full, then the randomer randomly picks one record in the buffer and releases it to the next component. Note that at any time point when one record is picked and released, it may be a real or dummy record. As a result, if a new record arrives the cloud at an improbable time point, the adversary cannot conclude with certainty whether it is dummy or not. Similarly, when the adversary does not see any record at an expected time point at the cloud, she/he cannot be sure it is removed or not due to the uncertainty caused by the randomer. The leakage caused by dealing naively with positive or negative noise (i.e., dummy records or removed true records) is thus addressed by the randomer. Moreover, FRESQUE does not require knowing in advance any data distribution.

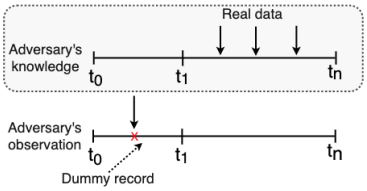


Figure 7: Randomer : Possible Issue of a Tiny Buffer

Challenges of randomer. One of the challenges of using randomer is how to choose a right size for its buffer. Intuitively, a large buffer gives high security. However, if the chosen size is too big, the system may confront bottlenecks at collector, particularly at the checking node. Otherwise, a tiny buffer may result above leak (the extreme case being a buffer of size 1). As an example depicted in Figure 7, we first assume that no real data is present during the period between t_0 and t_1 (the publishing time interval is $[t_0, t_n]$) and the size of the buffer is much smaller than the total number of dummy records. Since all dummy records are randomly released over a publishing time interval, it may happen that the buffer is full of dummy records, e.g., during the period $[t_0, t_1]$. The trigger function is thus activated and dummy records in the buffer are released before t_1. These dummy records will be recognized by the adversary who has prior knowledge about real data distribution. Fortunately, such situation only happens as the randomer buffer is much smaller than the total number of dummy records. Otherwise, if the randomer buffer is large enough, no record will appear at the cloud in that period. Therefore, the buffer size must be chosen to be sufficiently larger than the total number of dummy records of the publication.

A straightforward solution is to determine the buffer size by multiplying the actual number of the dummy records of a publication by several times. However, since we will publish the whole buffer at the end of the publishing time interval, the adversary may infer the size of the buffer, and hence the actual number of dummy records can be leaked. So the method of determining buffer size must (*) not depend on the real number of dummy records and (**) being larger than the number of the dummy records of a publication.

Note that since dummy records are generated due to the Laplace noise, the number of dummy records varies with each publication. It is thus difficult to choose a right capacity for the randomer buffer while meeting both (*) and (**). Fortunately, the noise in FRESQUE is sampled from the Laplace distribution, we can then choose buffer size based on the inverse CDF of the Laplace distribution with a very high probability, δ'. Intuitively, this approach gives an upper bound on the number of dummy records. Given a set of m leaves, denoted $L = \{l_1, ..., l_m\}$, we probabilistically compute the maximum number of dummy records of leaf l_i based on the inverse CDF of the Laplace distribution, considered as s_i. Then, $T = \sum_{i=1}^{m} s_i$ is viewed as the *maximum* number of dummy records of an index. To guarantee the buffer size is larger than T, we multiply it by a configurable coefficient, α. To ensure the buffer size is larger than the total number of dummy records, we suggest to set $\alpha \geq 2$. Then, the buffer size, S, of the randomer is: $S = \sum_{i=1}^{m} s_i \times \alpha$ (or $S = T \times \alpha$), where $\alpha \geq 2$.

Finally, the position of the randomer within the architecture of the collector is important as well: it must be put before the checker and the updater (so that it processes all records, including the removed ones) and after the parser and the encrypter (for obvious latency reasons).

5.3 Architecture of FRESQUE

Following the key design features in Section 5.1, we now detail our ingestion framework to support efficient range query processing over encrypted data. Especially, we describe the orchestration of different components in this architecture.

(a) Ingestion life cycle. The collector of FRESQUE runs on a small cluster of commodity machines (see Figure 6). At the collector, one (and only one) node runs the Dispatcher (D) and all worker nodes in the cluster run a Computing Node (CN) while the randomer, the checker, and the updater run on the same Checking node (C).

When new records arrive to the dispatcher, they are immediately sent to the computing nodes according to a round robbin approach. This approach is used for the sake of load balancing. The computing node first pre-processes incoming data to get pairs of <leaf offset, e-record>. These pairs are then sent to the checking node. After being randomized and checked at the checking node, such pairs are forwarded to the cloud or the merger. Note that the dispatcher, the computing node, the merger, and the checking node can coexist on the same node.

(b) Instantiation of FRESQUE. In order to demonstrate how data

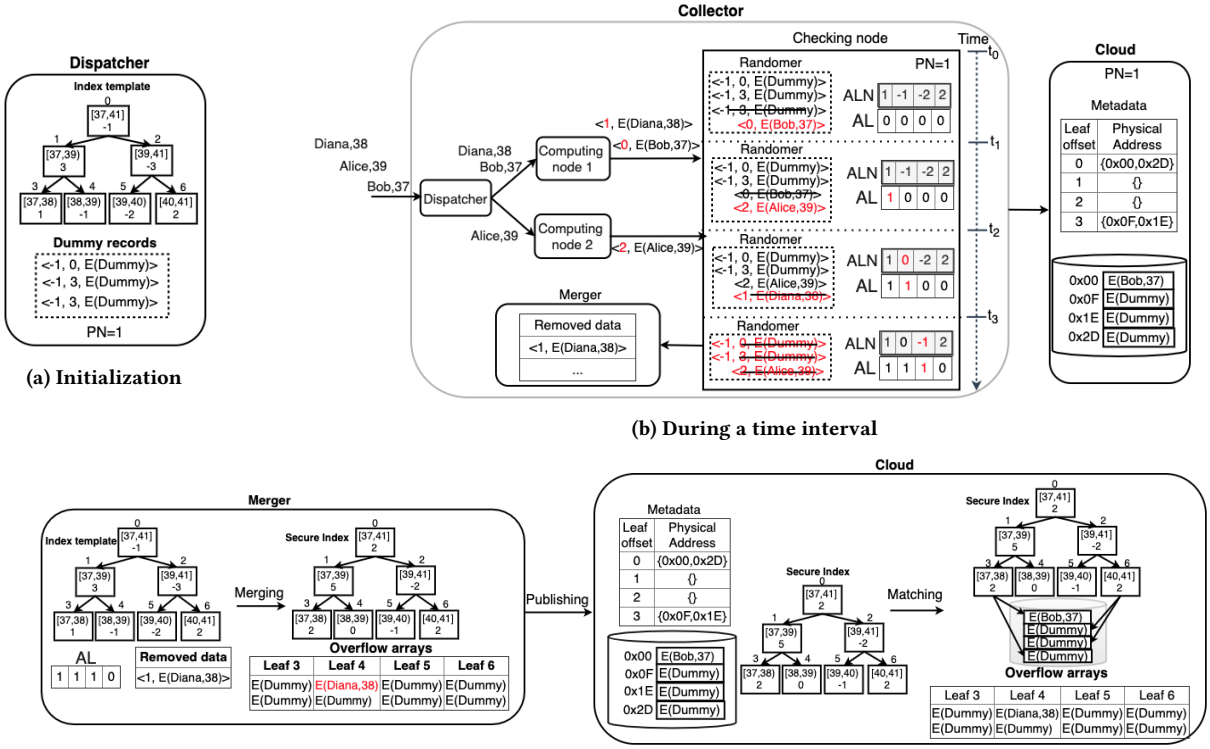


Figure 8: An example demonstrating how FRESQUE processes incoming data by using two computing nodes. Assume that the size of randomer buffer is 4 pairs of e-record.

is processed at the collector and transported to the cloud, Figure 6 shows the composition of FRESQUE running on five nodes at the collector and Figure 8 gives a running example.

Dispatcher (D): At the beginning of a publishing time interval, the dispatcher initiates an Index Template (IT), dummy records, and a Publication Number (PN), as illustrated in Figure 8a. The dispatcher then sends the IT, PN and all dummy records to the checking node. During a publishing time interval, whenever the dispatcher receives new data from data sources, it distributes the data to the computing nodes in a round-robin fashion. As an example shown in Figure 8b, there are three records, (Bob,37), (Alice,39), (Diana,38), arriving in order at the dispatcher. These records are then distributed to the two computing nodes. At the end of each publishing time interval, the dispatcher sends a *publishing* message to all available computing nodes and to the checker. By using the asynchronous publishing method, a new publication is immediately started after a *publishing* message is sent instead of waiting for the publishing tasks to be done. This allows the system to continuously ingest arrivals. By completely removing heavy jobs (e.g., parsing, encrypting, and checking) from the dispatcher and using the asynchronous publishing method, throughput ingestion is maximized at this component.

Computing Node (CN): During a publishing time interval, when new data comes, the computing node parses the raw data into a record, calculates the leaf offset, and encrypts it. Then, a pair of <leaf offset, e-record> is transferred to the checking node. As showed in Figure 8b, after passing the two computing nodes, three records are now parsed, encrypted, and associated with the corresponding leaf offsets, 0, 2, 1, respectively. When the computing node receives a *publishing* message from the dispatcher, it waits for a *done* message from the checking node. Notably, during the meantime, all incoming data will be processed and stored in local in-memory buffers at the computing nodes. By doing it, the delay of performing heavy tasks on buffered data is reduced when a new publication is started.

As mentioned earlier, the parser and the encrypter mainly cause throughput degradation in the system. With the parallel approach, the degradation is reduced significantly and only relies on the number of the computing nodes used. Interestingly, this approach not only allows to easily scale the throughput up, but also shortens the publishing time at the collector. For instance, PINED-RQ++ has to sequentially encrypt removed records and insert them into overflow arrays at the end of each publishing time interval, whereas they are now encrypted in a parallel manner by a set of networked machines during that period. As a result, at the end of each publishing time interval, the collector only randomly inserts removed encrypted records into the corresponding overflow arrays before transferring them to the cloud, reducing the publishing time in FRESQUE.

Checking node (C): At the beginning of a publishing time interval, the checking node receives Index Template (IT) and Publication Number (PN) from the dispatcher. It first initiates the corresponding AL and ALN (see Figure 8b). The checking node then forwards the IT to the merger while the PN is sent to the cloud. During a publishing time interval, when a pair of <leaf offset, e-record> arrives, it stores that pair in the buffer of the randomer. If the buffer is full, one of them is randomly picked and passed to the checker. Next, the checker gets its leaf offset (e.g., i) from the selected pair. If the i^{th} element of ALN is less than zero, the checker increases the value of the i^{th} element of both ALN and AL by one, and then sends that pair to the merger

as removed. Otherwise, that pair is sent to the updater, and only the value of the i^{th} element of AL is increased by one. Finally, that pair is sent to the cloud. As the example presented in Figure 8b, when the pair <0, (Bob,37)> comes to the checking node at timestamp t_0, it is inserted into the randomer's buffer. When this pair is released at timestamp t_1, the 0^{th} element of AL is increased by one since the 0^{th} element of ALN is positive. Otherwise, at timestamp t_2, when the pair <1, (Diana,38)> is considered, since it belongs to a negative element of ALN, the 1^{st} element of AL and ALN are both increased by one and this pair is then sent to the merger.

When the checking node receives *publishing* messages from all available computing nodes, it will send the updated AL to the merger (see Figure 8c). We emphasize that the condition of receiving publishing messages from all computing nodes needs to be guaranteed so that the consistency of publications is achieved. In other words, it makes sure that all (dummy/real) data of the current publication, that are sent by the dispatcher, are received by the checking node. The randomer buffer is then shuffled and published to the cloud. Finally, the checking node sends a *done* message back to the computing nodes.

It is worth noting that the checker and the updater will ignore the dummy records when they pass the checking node. This means that the counts of AL and ALN are independent of such dummy data. To achieve it, the checker and the updater need to perceive which incoming record is dummy in order to ignore it during the updating process. Nonetheless, the difficulty is that they all become ciphertexts after being encrypting by the computing nodes. To address it, we add to dummy records a *special* flag (e.g., -1) to distinguish them from real data. This straightforward technique allows the checker and the updater to know which record is dummy or real. As shown in Figure 8b, at timestamp t_0, a dummy pair is released by the checking node, and does not lead to any update on AL and ALN.

Even if all tasks at the checking node are designed to run sequentially such as the checker and the updater, they do not have much impact on the ingestion throughput at the collector. Moreover, thanks to the array representation, our architecture diminishes the complexity of the updating and checking tasks from $O(\log_k n)$ to $O(1)$, and hence shortening the delay of processing a record and boosting the consumption throughput.

Merger (M): At the beginning of each publishing time interval, the merger receives IT and PN from the checking node, then keeps them in memory. During a publishing time interval, the merger may receive removed records from the checker. Whenever the merger receives the updated AL, it triggers a new merging job that performs publishing tasks, e.g., combining IT and AL to achieve the complete secure index, generating overflow arrays (OAs) to conceal the removed records. Finally, the merger sends them to the cloud with the corresponding PN, as shown in Figure 8c.

Cloud: When the cloud receives a new PN from the checking node, it creates a new file for storing the incoming data. However, when its secure index is published by the merger, the published data will be read from the file on disk for a matching process and finally written back to disk again. Such approach gives rise to high I/O overhead. Instead, we keep small information about the published data, e.g., *metadata*, that is used for the matching process. Specifically, when a pair of <leaf offset, e-record> arrives, the cloud writes the e-record to disk, gets its physical address, and caches a pair of <leaf offset, physical location> in memory. To boost the matching process, we organize *metadata* in the form of <leaf offset, list of physical locations>, as demonstrated in Figure 8b. Such *metadata* is relatively small and independent of the size of e-records. When a publication comes from the merger, the matching process immediately associates the physical address of e-records with leaves based on the cached metadata. The metadata is finally destroyed (see Figure 8c).

(c) Query processing. In FRESQUE, when a query comes at the cloud, it is evaluated on both indexed and unindexed data. With regard to indexed data, the query processing strategy is applied as in Section 4.1. Meanwhile, unindexed data are processed one by one based on the query range. The (removed) records have a range overlapping the query range at the cloud, the randomer, and the merger are returned to the client.

6 SECURITY ANALYSIS

We develop FRESQUE that builds a PINED-RQ index [33] during a publishing time interval. With such an approach, at the end of each publishing time interval, all parts of the index (e.g., IT, AL, and removed e-records) are combined at the merger to get a secure index and overflow arrays. In other words, this process only occurs at the trusted collector, and hence FRESQUE apparently inherits the privacy protection level of the PINED-RQ index and trivially satisfies (ϵ, δ)-Probabilistic-SIM-CDP against offline attackers and (simple) online attackers.

The main difference between FRESQUE and PINED-RQ with respect to the index creation function is that FRESQUE publishes encrypted records immediately. Informed online attackers may be able to gain information about positive or negative noises based on the expected time distribution of incoming real records. However, thanks to the randomer, this leakage is mitigated. Theorem 2 claims the security of FRESQUE.

Theorem 2 (Security of FRESQUE*):* The index creation function of FRESQUE satisfies (ϵ, δ)-Probabilistic-SIM-CDP [33] against offline attackers and (simple) online attackers, and mitigates the information leak against informed online attackers.

Proof. *(Sketch)* We only consider informed online attackers because the two other attackers are trivial.

Considering dummy records (information leak about positive noises). First, we consider the case where a record arrives at the cloud at the time point at which there is real data. Since real/dummy records are randomly mixed together before being released, the adversary is unable to distinguish dummy records from real ones and does not obtain additional useful information about the values of the positive noises.

Second, we consider the case where a record arrives at the cloud at an unlikely time point at which there is no real data. This situation happens when a dummy record is inserted into a full randomer buffer. This means that with high probability the randomer buffer contains both real and dummy records. When receiving a record recently picked from the buffer, the cloud is thus unable to distinguish dummy records from real ones and does not learn additional information about the values of the positive noises.

Third, we consider the case where the checking node sends the full randomer buffer to the cloud at the end of a publishing time interval. Dummy records are mixed with real ones at the randomer during a publishing time interval. Additionally, the ratio between real and dummy data at any time point is hidden from the adversary (see Section 5.2), hence the adversary does not

obtain additional useful information about the values of the positive noises from this case.

Fourth, we consider the case where the randomer contains only dummy records and no real ones. Note that such situation only occurs if all dummy records are released before real data arrives at the collector. If the buffer of the randomer were allowed to be smaller than or equal to the total number of dummy records, a dummy record would be picked with certainty and released to the cloud. As a result, the adversary would learn with certainty that it is dummy. However, FRESQUE requires that the buffer of the randomer is chosen to be much larger than the total number of dummy records of a publication, with (tunable) high probability (see Section 5.2). This makes this case highly improbable.

Considering removed records (information leak about negative noises). Since dummy records will not be deleted by the checking node, we only focus on real records. Recall that the decision to remove a real record is taken by the checker, after the randomer, and that removed records are buffered by the merger in order to be published within the overflow arrays at the end of the publishing time interval. Without randomer, the artificial removal of records due to negative noise may impact the number of records sent to the cloud and thus, slightly, the time distribution of the records sent to the cloud. However, first, the additional dummy records mitigate the decrease in the number of records overall (recall that the Laplace distribution used for generating the noises is symmetric around 0), and second, with the randomer, the delay introduced by the randomer buffer in releasing both dummy and real records also impacts the time distribution of records to the cloud, similar to the removal of true records, which mitigates the information leak about the values of the negative noises. □

Comparison with PINED-RQ [33]. The highest security of FRESQUE is achieved when the coefficient α is chosen so that the randomer buffer can contain the whole dataset and all dummy records. In that case, at the end of each publishing time interval, the randomer shuffles and sends the buffer to the cloud along with a secure index and overflow arrays. It is easy to see that this is exactly the publishing process of PINED-RQ. Thus, in that case, FRESQUE has the same level of privacy against informed online attackers as PINED-RQ, and thus also satisfies exactly (ϵ, δ)-Probabilistic-SIM-CDP against all attackers.

7 EVALUATION

We evaluate FRESQUE against PINED-RQ++ due to its outperformance compared to other prior schemes (see Table 1). We mainly focus on the metrics contributing to the scalability of the system, namely ingestion throughput and publishing latency at the collector as well as at the cloud.

7.1 Benchmark Environment

Table 2: Experimental environment

Component	CPU (2.4 GHz)	Memory (GB)	Disk (GB)
Dispatcher	4	8	80
Merger	4	8	80
Checking node	4	8	80
Computing node	2	2	20
Data source	4	16	80
Cloud	16	64	160

We implemented FRESQUE in Java 1.8.0[3]. Data was encrypted by the Java package (javax.crypto). We ran our experiments on the Galactica platform [4] and organized FRESQUE as a cluster of 17 nodes, including 12 computing nodes, running on Ubuntu 14.04.4 LTS. Each node was used to run one component of FRESQUE. The configurations of nodes are detailed in Table 2. The TCP socket was used for exchanging data among the components of FRESQUE.

[3]The code is available at https://gitlab.inria.fr/vtran/fresque.git.

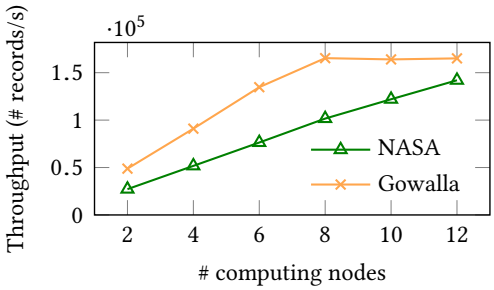


Figure 9: Ingestion throughput of FRESQUE

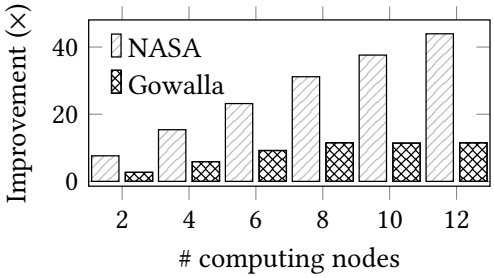


Figure 10: FRESQUE's improvement compared to PINED-RQ++

We evaluate our solution on two real datasets: NASA log [1] (1, 569, 898 records, five attributes) and Gowalla [22] (6, 442, 892 records, three attributes). We use the reply byte and check-in time as indexed attributes, respectively. Based on these datasets, the domain of the reply byte is divided into 3421 bins and each bin interval represents 1 KB. Meanwhile, the domain of the check-in time is 626 bins and each bin interval implies one hour. The fanout is set to 16. We use a publishing time interval of 60 seconds and incoming data rate is 200k records per second. The initial privacy budget and coefficient is set to 1 and 2, respectively, for all experiments unless otherwise stated. Both δ and δ' are set to 99% and every experiment was run over ten minutes. Then, we present the averaged results of ten publications in Section 7.2.

7.2 Results

Ingestion throughput. We first present the ingestion throughput of FRESQUE with a varied number of computing nodes. Then we compare its ingestion throughput to those of the (non-)parallel PINED-RQ++. The results in Figure 9 show that the throughput of FRESQUE significantly increases as the number of computing nodes goes up. Especially, the highest throughput is reached at ~142k records/second (NASA) and at ~165k records/second (Gowalla) with 12 and 8 computing nodes respectively. As compared to ArxRange [30], one of the state-of-the-art solutions, FRESQUE reaches an ingestion throughput that is at least two orders of magnitude higher.

(a) Comparison with non-parallel PINED-RQ++. With the given settings, non-parallel PINED-RQ++ is able to ingest only 3,159 records/s in NASA and 13,223 records/s in Gowalla. Such ingestion throughputs are substantially lower than those of FRESQUE. The results in Figure 10 demonstrate the outperformance of

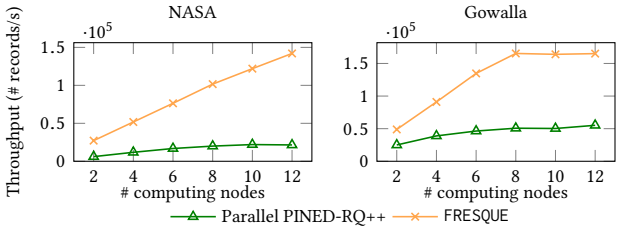


Figure 11: Comparison of ingestion throughput with parallel PINED-RQ++

FRESQUE compared to non-parallel PINED-RQ++. The enhancement goes up as the number of computing nodes grows. The highest improvement can be seen as the collector is configured as a 12-computing node cluster, and the ingestion throughput is improved by ~11× and ~43× in Gowalla and NASA dataset, respectively. Even if only two computing nodes are used, FRESQUE can achieve the improvement of 7.61× (NASA) and 2.69× (Gowalla). Compared to Gowalla, NASA always exhibits higher improvement with the same number of computing nodes. The major source of this gap comes from the fact that the record size and the domain of NASA record are larger than those of Gowalla. Based on such observation, we can conclude that FRESQUE would be more beneficial as datasets have larger size and/or domain.

(b) Comparison with parallel PINED-RQ++. The throughput of FRESQUE is always higher than that of parallel PINED-RQ++ as we vary the number of computing nodes at the collector, as shown in Figure 11. The setting of 12-computing node cluster gives the biggest gap, the throughput of FRESQUE is ~5.6× (NASA) and ~2.2× (Gowalla) better than that of parallel PINED-RQ++. Noted that since the throughput in FRESQUE reaches the peak as we use 8 computing nodes in Gowalla, the use of more computing nodes does not bring more benefit.

Throughput degradation. We measure the throughput degradation at the collector of the three prototypes. Such metric is obtained by comparing their maximum ingestion throughput with the maximum incoming throughput (without any processing on incoming data) at the collector. As shown in Figure 12, FRESQUE experiences the lowest throughput degradation among the three prototypes, with a reduction of at least ~3.9× (compared to parallel PINED-RQ++) in NASA, and at most ~7.9× (compared to PINED-RQ++) in Gowalla.

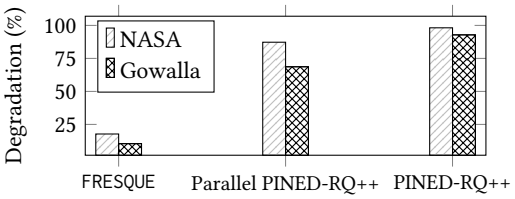


Figure 12: Throughput degradation at the collector

Publishing time. We now turn our attention to the publishing time metric, i.e., the time required to publish a dataset with FRESQUE and with parallel PINED-RQ++. Noted that FRESQUE consists of the three main components, namely the dispatcher, the checking node, and the merger which mainly decide the publishing time at the collector. We thus measure the delay of the three components separately. Additionally, we consider the time needed to perform a matching process at the cloud. This is because a long delay of this process might also lead to bottlenecks.

(a) Publishing time at the dispatcher. As shown in Figure 13, the time is always lower than 520ms with NASA and 200ms with

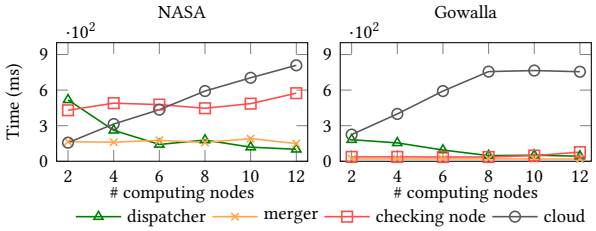


Figure 13: Publishing time in FRESQUE

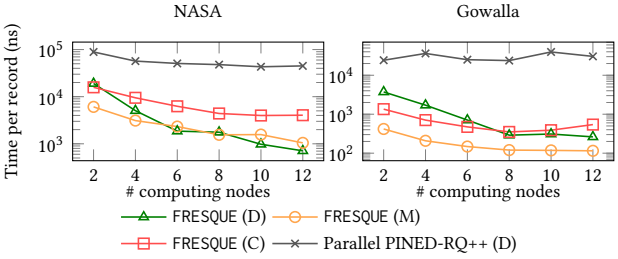


Figure 14: Comparison of publishing time at collector with parallel PINED-RQ++

Gowalla. The delay even gradually decreases as the number of computing nodes increases. In particular, the dispatcher takes only 101ms (NASA) and 19ms (Gowalla) for performing the publishing tasks in a 12-computing node cluster.

(b) Publishing time at the merger. The results in Figure 13 indicates that the time is virtually unchanged in the two datasets as their size changes. Specifically, the time with NASA fluctuates between 149ms and 191ms while that with Gowalla varies between 18ms and 20ms. Since the domain size of NASA (3421 bins) is larger than the one of Gowalla (626 bins), the NASA experiences a higher publishing time than that of the Gowalla dataset.

(c) Publishing time at the checking node. In this study, we attempt to design FRESQUE so that the checking node has a lightweight publishing job and has a reduced impact on the ingestion performance. In particular, the checking node only sends the buffer of the randomer to the cloud and the updated AL to the merger at the end of each publishing time interval. The results in Figure 13 show that the time is under 600ms with NASA and 80ms with Gowalla. It can be understood that the publishing time at the checking node is mainly represented by the time of sending the randomer buffer that varies according to the required level of security. A huge randomer buffer results in long publishing time at this component. Fortunately, since the computing nodes always process and cache incoming data during the meantime, the impact on the ingestion throughput is negligible. We will evaluate the randomer below in the latest part of this section.

(d) Matching time at the cloud. To show the efficiency of FRESQUE at the cloud side, we measure the time required to associate *metadata* (physical locations of records) with published index. As depicted in Figure 13, the time in FRESQUE goes up according to data size. Nonetheless, FRESQUE spends only 877ms and 837ms on matching the large dataset of 8.1M records (NASA) and 9.8M records (Gowalla), respectively. These performances come from the deletion of the matching table from FRESQUE's architecture.

(e) Comparison with parallel PINED-RQ++. We now compare publishing time at the collector between FRESQUE and parallel PINED-RQ++. Since the different numbers of computing nodes used result in different publication sizes, we consider the time is required to publish a record instead of a whole dataset. The results in Figure 14 show that parallel PINED-RQ++ (dispatcher) takes longer delay than FRESQUE (dispatcher, checking node, and merger) for

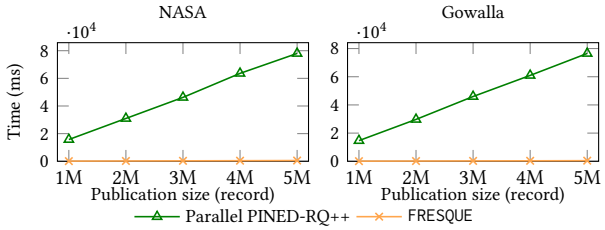


Figure 15: Comparison of matching time at cloud with parallel PINED-RQ++

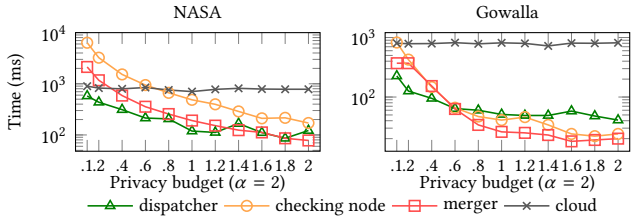


Figure 16: Publishing time with different privacy budgets

the two datasets used. Regarding the dispatcher, the publishing time of FRESQUE is at most ~62× and ~127× lower with NASA and Gowalla, respectively, compared to parallel PINED-RQ++.

Matching time. We also evaluate the matching time needed to process a publication between parallel PINED-RQ++ and FRESQUE. The results in Figure 15 show that the time of parallel PINED-RQ++ increases when publications are larger. For example, when a dataset of 5M records is used, the matching time in parallel PINED-RQ++ reaches ~78s (NASA) and ~76s (Gowalla). In contrast, FRESQUE constantly maintains a short time for processing a publication at the cloud, with a maximum is ~54ms (NASA) and ~43ms (Gowalla). The matching time of FRESQUE is at least two orders of magnitude shorter than that of parallel PINED-RQ++.

Impact of the randomer. For mitigating the information disclosed to the informed online attacker, the randomer maintains a local buffer for perturbing incoming data. A large buffer may introduce a bottleneck at the collector. We thus evaluate the impact of this component here. Indeed, the buffer size is mainly determined by two configurable parameters, namely privacy budget ϵ and coefficient α. Hence, we run various experiments with varied values of the two parameters to evaluate the impact of the randomer. We use a configuration of 10 computing nodes.

(a) Privacy budget ϵ. We now consider the impact of the randomer in terms of publishing time as we use different privacy budgets, ranging from 0.1 to 2.0, for a publication. In these experiments, we record the publishing time at the collector (dispatcher, checking node, and merger), and the matching time at the cloud. The results in Figure 16 show that the privacy budget influences the publishing time at the three components. Indeed, as a smaller privacy budget is used, their publishing time goes up. The highest increase is witnessed at the checking node, approximately 7s (NASA) and about 0.8s (Gowalla) for the budget of 0.1. Similarly, as the privacy budget declines, the size of overflow arrays and the number of dummy/removed records go up, causing a slight increase of the publishing time at the dispatcher and the merger.

(b) Coefficient α. We adjust the value of α to see the impact of randomer on publishing time at the checking node, the merger and the cloud. As expected, when we increase the value of α, the publishing time grows (see Figure 17). However, even if α is set to 20, the checking node only takes about 6s (NASA) and 0.8s (Gowalla). Also, the time does not change much at the dispatcher,

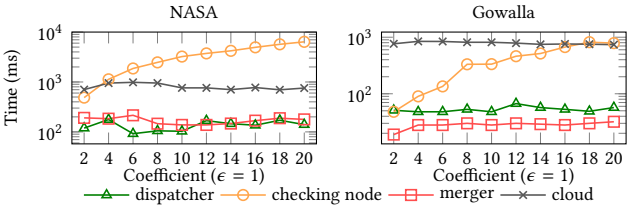
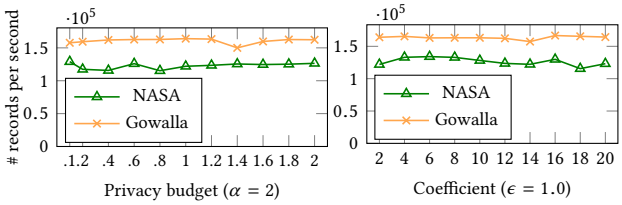


Figure 17: Publishing time with different coefficients



(a) Varying privacy budgets **(b) Varying coefficients**

Figure 18: Ingestion throughput of FRESQUE with randomer

the merger and the cloud.

(c) Impact of the randomer on ingestion throughput. We also consider the ingestion throughput at the collector as we vary the two parameters ϵ and α. Although the publishing time at the checking node goes up as we use smaller privacy budget and/or larger coefficient, the ingestion throughput at the collector is relatively stable. This is because while the checking node prepares publish the current dataset, including the sending of randomer to the cloud, incoming data of the new publication is still processed and buffered at the computing nodes. As it can be seen in Figure 18a and Figure 18b, the results show that the throughput with the NASA dataset fluctuates between ~115k records/s and ~134k records/s while that of Gowalla ranges from ~150k records/s to ~166k records/s.

8 DISCUSSION

We present a possible real-life application of FRESQUE based on the FluTracking use-case [3].

Flutracking is a web-based survey of influenza-like illness. This system weekly sends a link via email to all participants who will then submit required information via a web interface. The submitted data can be managed in a cloud and accessed by authorized users for analysis and prediction.

Although our description of FRESQUE focuses on the insertion of one record per individual, it is simple to extend our approach to the case of multiple records per individual. For example, in Flutracking [3], an individual can submit personal data several times to the database, at most once for a week. For such case, an important question is how to manage privacy budget over multiple insertions of the same individual.

In the targeted use case, it is unlikely to have multiple records of the same individual over a short period (e.g., weekly). Therefore, we can assume that a dataset of each period (e.g., week) is published with a secure index, and this publication consists of at most one record per individual. For each dataset, the system uses a portion of the total privacy budget ϵ_{total} for constructing a secure index. To determine how much budget is spent for a publication, an admin may necessarily determine how long the system needs secure indices for fast range query processing. ϵ_{total} is then divided according to the determined period. For instance, if the system must maintain indices for one year (52 weeks), then an admin can divide the total privacy budget ϵ_{total} into 52 equal

portions, $\epsilon_1, ..., \epsilon_{52}$, so that $\epsilon_{total} = \sum_{i=1}^{52} \epsilon_i$. Each of which is used to publish dataset of one week. Certainly, the system needs to make sure that an individual contributes at most one record per publication. Fortunately, thanks to the existing collecting method of the Flutracking, this work can simply be achieved. In particular, a unique link can be sent to all participants every Monday. The link is set to expired and a dataset is published before the next Monday. This ensures that a participant links to at most one record per publication.

9 CONCLUSION

This paper presents FRESQUE, a scalable ingestion framework for secure range query processing over encrypted data on clouds. We thoroughly analyze and identify the problems of the-state-of-the-art solutions related to the degradation of the ingestion throughput, with a special focus on PINED-RQ++. To address these drawbacks, we design a new architecture that is fully distributed at the collector. Additionally, we introduce a data representation as well as an asynchronous publication mechanism. All of them together allows FRESQUE to achieve intensive consumption throughput, reaching over 160K records/s. Moreover, we introduce and carefully integrate the randomer into our new architecture to improve the practicality and security of FRESQUE as compared to PINED-RQ++. We formally analyse the security guarantees of FRESQUE. Lastly, we discuss a potential application of FRESQUE based on a real-life example. Future works include coping with multiple records per individual and designing alternative indexes based on well-known highly concurrent data structures (e.g., Masstree [25] and ART [21]).

ACKNOWLEDGMENTS

This work was partially funded by LTC and Pôle d'Excellence Cyber.

REFERENCES

[1] NASA Log. http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html (1996).
[2] Flu Near You. https://flunearyou.org/ (2020).
[3] FluTracking.net. http://info.flutracking.net/ (2020).
[4] Galactica. https://galactica.isima.fr/index.html (2020).
[5] Agrawal R., Kiernan J., Srikant R., and Xu Y. Order Preserving Encryption for Numeric Data. In: SIGMOD '04, 563–574 (2004).
[6] Boldyreva A., Chenette N., Lee Y., and O'Neill A. Order-Preserving Symmetric Encryption. In: A. Joux, ed., Advances in Cryptology - EUROCRYPT 2009, 224–241. Springer Berlin Heidelberg, Berlin, Heidelberg (2009).
[7] Boldyreva A., Chenette N., and O'Neill A. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In: P. Rogaway, ed., Advances in Cryptology – CRYPTO 2011, 578–595 (2011).
[8] Boneh D. and Waters B. Conjunctive, Subset, and Range Queries on Encrypted Data. In: TCC'07, 535–554 (2007).
[9] Dalton C., Carlson S., Butler M., Cassano D., Clarke S., Fejsa J., and Durrheim D. Insights From Flutracking: Thirteen Tips to Growing a Web-Based Participatory Surveillance System. JMIR Public Health Surveill, 3(3):e48 (2017).
[10] Demertzis I., Papadopoulos S., Papapetrou O., Deligiannakis A., Garofalakis M., and Papamanthou C. Practical Private Range Search in Depth. ACM Trans. Database Syst., 43(1):2:1–2:52 (2018).
[11] Dwork C. Differential Privacy. In: ICALP'06, 1–12 (2006).
[12] Dwork C., McSherry F., Nissim K., and Smith A. Calibrating Noise to Sensitivity in Private Data Analysis. In: TCC'06, 265–284 (2006).
[13] Dwork C. and Roth A. The Algorithmic Foundations of Differential Privacy. Found. Trends Theor. Comput. Sci., 9(3-4):211–407 (2014).
[14] Elmore A.J., Das S., Agrawal D., and El Abbadi A. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In: SIGMOD '11, 301–312 (2011).
[15] Goldreich O. Foundations of Cryptography: Volume 2, Basic Applications. Cambridge University Press, New York, NY, USA (2004).
[16] Guerrisi C., Ecollan M., Souty C., Rossignol L., Turbelin C., Debin M., Goronflot T., Boëlle P.Y., Hanslik T., Colizza V., and Blanchon T. Factors associated with influenza-like-illness: a crowdsourced cohort study from 2012/13 to 2017/18. BMC Public Health, 19(1):879 (2019).
[17] Hacigümüş H., Iyer B., Li C., and Mehrotra S. Executing SQL over Encrypted Data in the Database-service-provider Model. In: SIGMOD '02, 216–227 (2002).
[18] Help-Net-Security. 52% of companies use cloud services that have experienced a breach. https://www.helpnetsecurity.com/2020/01/28/accessing-cloud-services/ (2020).
[19] Hore B., Mehrotra S., Canim M., and Kantarcioglu M. Secure Multidimensional Range Queries over Outsourced Data. The VLDB Journal, 21(3):333–358 (2012).
[20] Hore B., Mehrotra S., and Tsudik G. A Privacy-preserving Index for Range Queries. In: VLDB '04, 720–731 (2004).
[21] Leis V., Kemper A., and Neumann T. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In: ICDE '13, 38–49. IEEE Computer Society (2013).
[22] Leskovec J. and Krevl A. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data (2014).
[23] Li R. and Liu A.X. Adaptively Secure Conjunctive Query Processing over Encrypted Data for Cloud Computing. In: ICDE'17, 697–708 (2017).
[24] Li R., Liu A.X., Wang A.L., and Bruhadeshwar B. Fast Range Query Processing with Strong Privacy Protection for Cloud Computing. Proc. VLDB Endow., 7(14):1953–1964 (2014).
[25] Mao Y., Kohler E., and Morris R.T. Cache Craftiness for Fast Multicore Key-Value Storage. In: EuroSys '12, 183–196. ACM (2012).
[26] Mavroforakis C., Chenette N., O'Neill A., Kollios G., and Canetti R. Modular Order-Preserving Encryption, Revisited. In: SIGMOD '15, 763–777 (2015).
[27] McSherry F.D. Privacy Integrated Queries: An Extensible Platform for Privacy-preserving Data Analysis. In: SIGMOD '09, 19–30 (2009).
[28] Mironov I., Pandey O., Reingold O., and Vadhan S. Computational Differential Privacy. In: S. Halevi, ed., Advances in Cryptology - CRYPTO 2009, 126–142 (2009).
[29] Papadimitriou A., Bhagwan R., Chandran N., Ramjee R., Haeberlen A., Singh H., Modi A., and Badrinarayanan S. Big Data Analytics over Encrypted Datasets with Seabed. In: OSDI '16, 587–602 (2016).
[30] Poddar R., Boelter T., and Popa R.A. Arx: An Encrypted Database Using Semantically Secure Encryption. Proc. VLDB Endow., 12(11):1664–1678 (2019).
[31] Popa R.A., Li F.H., and Zeldovich N. An Ideal-Security Protocol for Order-Preserving Encoding. In: SP '13, 463–477 (2013).
[32] Popa R.A., Redfield C.M.S., Zeldovich N., and Balakrishnan H. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In: SOSP '11, 85–100 (2011).
[33] Sahin C., Allard T., Akbarinia R., El Abbadi A., and Pacitti E. A Differentially Private Index for Range Query Processing in Clouds. In: ICDE '18, 857–868 (2018).
[34] Tran H.V., Allard T., D'Orazio L., and Abbadi A.E. Range Query Processing for Monitoring Applications over Untrustworthy Clouds. In: EDBT'19, 666–669 (2019).
[35] Tu S., Kaashoek M.F., Madden S., and Zeldovich N. Processing Analytical Queries over Encrypted Data. Proc. VLDB Endow., 6(5):289–300 (2013).
[36] Wen M., Lu R., Zhang K., Lei J., Liang X., and Shen X. PaRQ: A Privacy-Preserving Range Query Scheme Over Encrypted Metering Data for Smart Grid. IEEE Transactions on Emerging Topics in Computing, 1(1):178–191 (2013).

Cache on Track (CoT): Decentralized Elastic Caches for Cloud Environments

Victor Zakhary, Lawrence Lim, Divyakant Agrawal, Amr El Abbadi
UC Santa Barbara
Santa Barbara, California
victorzakhary,lawrenceklim,divyagrawal,elabbadi@ucsb.edu

ABSTRACT

Distributed caches are widely deployed to serve social networks and web applications at billion-user scales. This paper presents *Cache-on-Track* (CoT), a decentralized, elastic, and predictive caching framework for cloud environments. CoT proposes a new cache replacement policy specifically tailored for small front-end caches that serve skewed workloads with small update percentage. Small front-end caches are mainly used to mitigate the load-imbalance across servers in the distributed caching layer. Front-end servers use a heavy hitter tracking algorithm to continuously track the top-k hot keys. CoT dynamically caches the top-C hot keys out of the tracked keys. CoT's main advantage over other replacement policies is its ability to dynamically adapt its tracker and cache sizes in response to workload distribution changes. Our experiments show that CoT's replacement policy consistently outperforms the hit-rates of LRU, LFU, and ARC for the same cache size on different skewed workloads. Also, CoT slightly outperforms the hit-rate of LRU-2 when both policies are configured with the same tracking (history) size. CoT achieves server size load-balance with 50% to 93.75% less front-end cache in comparison to other replacement policies. Finally, experiments show that CoT's resizing algorithm successfully auto-configures the tracker and cache sizes to achieve back-end load-balance in the presence of workload distribution changes.

1 INTRODUCTION

Social networks, the web, and mobile applications have attracted hundreds of millions of users who need to be served in timely personalized way [9]. To enable this real-time experience, the underlying storage systems have to provide efficient, scalable, and highly available access to big data.

Figure 1 presents a typical web and social network system deployment [9] where user-data is stored in a distributed back-end storage layer in the cloud. The back-end storage layer consists of a distributed in-memory caching layer deployed on top of a distributed persistent storage layer. The caching layer aims to improve the request latency and system throughput and to alleviate the load on the persistent storage layer at scale [44]. Distributed caching systems such as Memcached [3] and Redis [4] are widely adopted by cloud service providers such as Amazon ElastiCache [1] and Azure Redis Cache [2]. As shown in Figure 1, hundreds of millions of end-users send streams of page-load and page-update requests to thousands of stateless front-end servers. These front-end servers are either deployed in the same core datacenter as the back-end storage layer or distributed among other core and edge datacenters near end-users. Each end-user request results in hundreds of data object lookups and updates served from the back-end storage layer. According to Facebook

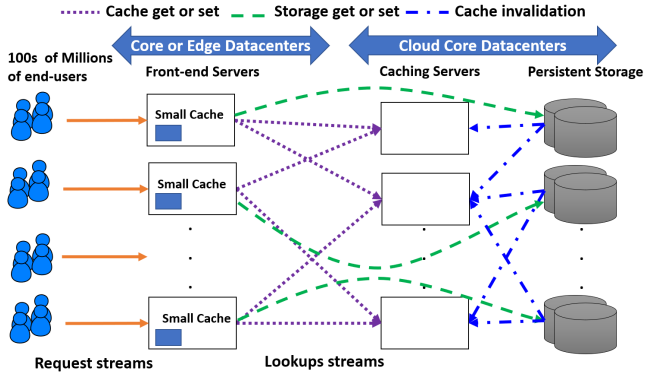


Figure 1: A typical web and social network system deployment

Tao [9], 99.8% of the accesses are reads and 0.2% of them are writes. Therefore, the storage system has to be **read optimized** to efficiently handle end-user requests at scale.

Redis and Memcached use consistent hashing [30] to distribute keys among several caching servers. Although consistent hashing ensures a fair distribution of the number of keys assigned to each caching shard, it does not consider the workload per key in the assignment process. Real-world workloads are typically skewed with few keys being significantly hotter than other keys [25]. This skew causes load-imbalance among caching servers.

Load imbalance in the caching layer can have significant impact on the overall application performance. In particular, it may cause drastic increases in the latency of operations at the tail end of the access frequency distribution [24]. In addition, the average throughput decreases and the average latency increases when the workload skew increases [11]. This increase in the average and tail latency is amplified for real workloads when operations are executed in chains of dependent data objects. A single page-load results in retrieving hundreds of objects in multiple rounds of data fetching operations [9, 38]. Finally, solutions that equally overprovision the caching layer resources to handle the most loaded caching server suffer from resource under-utilization in the least loaded caching servers.

In this paper, we propose Cache-on-Track (CoT); a decentralized, elastic, and predictive heavy hitter caching at front-end servers. CoT proposes a new cache replacement policy specifically tailored for small front-end caches that serve skewed workloads with small update percentage. CoT uses a small front-end cache to solve back-end load-imbalance as introduced in [20]. However, CoT does not assume perfect caching at the front-end. CoT uses the space saving algorithm [37] to track the top-k *heavy hitters*. The tracking information allows CoT to *cache* the exact top-C hot keys out of the approximate top-k tracked keys preventing cold and noisy keys from the long tail to replace hot keys in the cache. CoT is decentralized in the sense that each front-end independently determines its hot key set based on the key access

distribution served at this specific front-end. This allows CoT to address back-end load-imbalance without introducing single points of failure or bottlenecks that typically come with centralized solutions. In addition, this allows CoT to scale to thousands of front-end servers, a common requirement of social network and modern web applications. Unlike traditional replacement policies, CoT is elastic in the sense that each front-end uses its local load information to monitor its contribution to the back-end load-imbalance. Each front-end elastically adjusts its tracker and cache sizes to reduce the load-imbalance caused by this front-end. In the presence of workload changes, CoT dynamically adjusts front-end tracker to cache ratio in addition to both the tracker and cache sizes to eliminate any back-end load-imbalance.

In traditional architectures, memory sizes are static and caching algorithms strive to achieve the best usage of all the available resources. However, in cloud settings where there are theoretically infinite memory and processing resources and cloud instance migration is the norm, cloud end-users aim to achieve their SLOs while reducing the required cloud resources and thus decreasing their monetary deployment costs. CoT's main goal is to reduce the necessary front-end cache size *independently* at each front-end to eliminate server-side load-imbalance. In addition, CoT strives to dynamically find this minimum front-end cache size as the workload distribution changes. Reducing front-end cache size is crucial for the following reasons: 1) it reduces the monetary cost of deploying front-end caches. For this, we quote David Lomet in his recent works [33–35] where he shows that cost/performance is usually more important than sheer performance: *"the argument here is not that there is insufficient main memory to hold the data, but that there is a less costly way to manage data."*. 2) In the presence of data updates and when data consistency is a requirement, increasing front-end cache sizes significantly increases the cost of the data consistency management technique. Note that social networks and modern web applications run on thousands of front-end servers. Increasing front-end cache size not only multiplies the cost of deploying bigger cache by the number of front-end servers, but also increases several costs in the consistency management pipeline including a) the cost of tracking key incarnations in different front-end servers and b) the network and processing costs to propagate updates to front-end servers. 3) Since the workload is skewed, our experiments clearly demonstrate that the relative benefit of adding more front-end cache-lines, measured by the average cache-hits per cache-line and back-end load-imbalance reduction, drastically decreases as front-end cache sizes increase.

CoT's resizing algorithm dynamically increases or decreases front-end allocated memory in response to dynamic workload changes. CoT's dynamic resizing algorithm is valuable in different cloud settings where all front-end servers are deployed in the same datacenter or in different datacenters at the *edge*. These front-end servers obtain workloads of the same dynamically evolving distributions or different distributions. In particular, CoT aims to capture local trends from each individual front-end server perspective. In social network applications, front-end servers that serve different geographical regions might experience different key access distributions and different local trends (e.g., #miami vs. #ny).

We summarize our contributions in this paper as follows.

- Design and implement Cache-on-Track (CoT), a front-end cache replacement policy specifically tailored for small caches that serve skewed workloads with small update percentages.
- Design and implement CoT's resizing algorithm. The resizing algorithm dynamically minimizes the required front-end cache size to achieve back-end load-balance. CoT's built-in elasticity is a key novel advantage over other replacement policies.
- Evaluate CoT's replacement policy hit-rates to the hit-rates of traditional as well as state-of-the-art replacement policies, namely, LFU, LRU, ARC, and LRU-2. In addition, experimentally show that CoT achieves server size load-balance for different workload with **50% to 93.75%** less front-end cache in comparison to other replacement policies.
- Experimentally evaluate CoT's resizing algorithm showing that CoT successfully adjust its tracker and cache sizes in response to workload distribution changes.
- Report a *bug* at YCSB's [15] ScrambledZipfian workload generator. This generator generates workloads that are significantly less-skewed than the promised Zipfian distribution.

The rest of the paper is organized as follows. The related work is discussed in Section 2. In Section 3, the data model is explained. Section 4 further motivates CoT by presenting the main advantages and limitations of using LRU, LFU, ARC, and LRU-k caches at the front-end. We present the details of CoT in Section 5. In Section 6, we evaluate the performance and the overhead of CoT and the paper is concluded in Section 7.

2 RELATED WORK

Distributed caches are widely deployed to serve social networks and the web at scale [9, 38, 44]. Load-imbalancing among caching servers negatively affects the overall performance of the caching layer. Therefore, significant research has addressed the load-imbalacing problem from different angles. Solutions use different load-monitoring techniques (e.g., centralized tracking [6, 7, 26, 43], server-side tracking [11, 24], and client-side tracking [20, 28]). Based on the load-monitoring, different solutions redistribute keys among caching servers at different granularities. The following summarizes the related works under different categories.

Centralized load-monitoring: Slicer [7] and Centrifuge [6] are examples of centralized load-monitoring where a centralized control plane is separated from the data plane. Centrifuge uses consistent hashing to map keys to servers. However, Slicer propagates the key assignments from the control plane to the front-end servers. Slicer's control plane collects metadata about shard accesses and server workload. The control plane periodically runs an optimization algorithm that decides to redistribute, repartition, or replicate slices of the key space to achieve better back-end load-balance. Also, Slicer replicates the centralized control plane to achieve high availability and to solve the fault-tolerance problem in both Centrifuge [6] and in [11]. CoT is complementary to systems like Slicer and Centrifuge since CoT operates on a *fine-grain key* level at front-end servers while solutions like Slicer [7] operate on coarser grain slices or shards at the caching servers. Our goal is to cache heavy hitters at front-end servers to reduce key skew at back-end caching servers and hence, reduce Slicer's initiated re-configurations. Also, CoT is distributed and front-end driven that does not require any system component to develop a

global view of the workload. This allows CoT to scale to thousands of front-end servers without introducing any centralized points of failure or bottlenecks.

Server side load-monitoring: Another approach to load-monitoring is to distribute the load-monitoring among the caching shard servers. In [24], each caching server tracks its own hot-spots. When the hotness of a key surpasses a certain threshold, this key is replicated to γ caching servers and the replication decision is broadcast to all the front-end servers. Any further accesses on this hot key shall be equally distributed among these γ servers. Cheng et al. [11] extend the work in [24] to allow moving coarse-grain key cachelets (shards) among threads and caching servers. Our approach reduces the need for server side load-monitoring. Instead, load-monitoring happens at the edge. This allows individual front-end servers to independently identify their local trends and cache them without adding the monitoring overhead to the caching layer, a critical layer for the performance of the overall system.

Client side load-monitoring: Fan et al. [20] use a distributed front-end load-monitoring approach. This approach shows that adding a small perfect cache in the front-end servers has significant impact on solving the back-end load-imbalance. Fan et al. *theoretically* show through analysis and simulation that a small *perfect cache* at each front-end solves the back-end load-imbalance problem. Following [20], Gavrielatos et al. [21] propose *symmetric caching* to track and cache the hot-most items at every front-end server. Symmetric caching assumes that all front-end servers obtain the same access distribution and hence statically allocates the same cache size to all front-end servers. However, different front-end servers might serve different geographical regions and therefore observe different access distributions. CoT discovers the workload access distribution independently at each front-end server and adjusts the cache size to achieve some target load-balance among caching servers. NetCache [28] uses programmable switches to implement heavy hitter tracking and caching at the network level. Like symmetric caching, NetCache assumes a fixed cache size for different access distributions. To the best of our knowledge, CoT is the first front-end caching algorithm that exploits the cloud elasticity allowing each front-end server to independently reduce the necessary required front-end cache memory to achieve back-end load-balance.

Other works in the literature focus on maximizing cache hit rates for fixed memory sizes. Cidon et al. [12, 13] redistribute available memory among memory slabs to maximize memory utilization and reduce cache miss rates. Fan et al. [19] use cuckoo hashing [41] to increase memory utilization. Lim et al. [32] increase memory locality by assigning requests that access the same data item to the same CPU. Bechmann et al. [8] propose Least Hit Density (LHD), a new cache replacement policy. LHD predicts the expected hit density of each object and evicts the object with the lowest hit density. LHD aims to evict objects that contribute low hit rates with respect to the cache space they occupy. Unlike these works, CoT does not assume a static cache size. In contrast, CoT maximizes the hit rate of the available cache and exploits the cloud elasticity allowing front-end servers to independently expand or shrink their cache memory sizes as needed.

3 DATA MODEL

We assume a typical key/value store interface between the front-end servers and the storage layer. The API consists of the following calls:

- *v = get(k)* retrieves value *v* corresponding to key *k*.
- *set(k, v)* assigns value *v* to key *k*. *set(k, null)* to delete *k*.

Front-end servers use *consistent hashing* [30] to locate keys in the caching layer. Consistent hashing solves the *key discovery* problem and reduces key churn when a caching server is added to or removed from the caching layer. We extend this model by adding an additional layer in the cache hierarchy. As shown in Figure 1, each front-end server maintains a small cache of its hot keys. This cache is populated according to the accesses that are served by each front-end server.

We assume a client driven caching protocol similar to the protocol implemented by **Memcached** [3]. A cache client library is deployed in the front-end servers. *Get* requests are initially attempted to be served from the local cache. If the requested key is in the local cache, the *value* is returned and the request is marked as served. Otherwise, a *null* value is returned and the front-end has to request this key from the caching layer **at the back-end storage layer**. If the key is cached in the caching layer, its value is returned to the front-end. Otherwise, a *null* value is returned and the front-end has to request this key from the persistent storage layer and upon receiving the corresponding value, the front-end inserts the value in its front-end local cache and in the server-side caching layer as well. As in [38], a *set*, or an update, request invalidates the key in both the local cache and the caching layer. Updates are directly sent to the persistent storage, local values are set to null, and delete requests are sent to the caching layer to invalidate the updated keys. The Memcached client driven approach allows the deployment of a *stateless* caching layer. As requests are driven by the client, a caching server does not need to maintain the state of any request. This simplifies scaling and tolerating failures at the caching layer. Although, we adopt the Memcached client driven request handling protocol, our model works as well with write-through request handling protocols.

Our model is not tied to any replica consistency model. Each key can have multiple incarnations in the storage layer and the caching layer. Updates can be synchronously propagated if *strong consistency* guarantees are needed or asynchronously propagated if *weak consistency* guarantees suffice. Since the assumed workload is mostly read requests with very few update requests, we do not address consistency of updates in this paper. Achieving strong consistency guarantees among replicas of the same object has been widely studied in [11, 24]. Ghandeharizadeh et al. [22, 23] propose several complementary techniques to CoT to deal with consistency in the presence of updates and configuration changes. These techniques can be adopted in our model according to the application requirements. We understand that deploying an additional vertical layer of cache increases potential data inconsistencies and hence increases update propagation and synchronization overheads. Therefore, our goal in this paper is to reduce the front-end cache size in order to limit the inconsistencies and the synchronization overheads that result from deploying front-end caches, while maximizing their benefits on back-end load-imbalance.

4 FRONT-END CACHE ALTERNATIVES

Fan et al. [20] show that a small perfect cache in the front-end servers has big impact on the caching layer load-balance. A **perfect cache** of C cache-lines is defined such that accesses to the C hot-most keys always hit the cache while accesses to any other keys always miss the cache. The perfect caching assumption is impractical especially for dynamically changing and evolving workloads. Several replacement policies have been developed to

approximate perfect caching for different workloads. This section discusses the workload assumptions and various client caching objectives. This is followed by a discussion of the advantages and limitations of common caching replacement policies.

Workload assumptions: Real-world workloads are typically skewed with few keys being significantly hotter than other keys. In this paper, we assume skewed mostly read workloads with periods of stability (where hot keys remain hot during these periods).

Client caching objectives: Front-end servers construct their perspective of the key hotness distribution based on the requests they serve. Front-end servers aim to achieve the following caching objectives:

- The cache replacement policy should prevent cold keys from replacing hotter keys in the cache.
- Front-end caches should adapt to the changes in the workload. In particular, front-end servers should have a way to retire hot keys that are no longer accessed. In addition, front-end caches should have a mechanism to expand or shrink their local caches in response to changes in workload distribution. For example, front-end servers that serve uniform access distributions should dynamically shrink their cache size to *zero* since caching is of no value in this situation. On the other hand, front-end servers that serve highly skewed Zipfian (e.g., s = 1.5) should dynamically expand their cache size to capture all the hot keys that cause load-imbalance among the back-end caching servers.

A popular policy for implementing client caching is the LRU replacement policy. Least Recently Used (LRU) costs O(1) per access and caches keys based on their recency of access. This may allow cold keys that are recently accessed to replace hotter cached keys. Also, LRU cannot distinguish well between frequently and infrequently accessed keys [31]. Alternatively, Least Frequently Used (LFU) can be used as a replacement policy. LFU costs $O(log(C))$ per access where C is the cache size. LFU is typically implemented using a min-heap and allows cold keys to replace hotter keys at the top levels of the heap. Also, LFU cannot distinguish between old references and recent ones. This means that LFU cannot adapt to changes in workload. Both LRU and LFU are limited in their knowledge to the content of the cache and cannot develop a wider perspective about the hotness distribution outside of their static cache size.

Adaptive Replacement Cache (ARC) [36] tries to realize the benefits of both LRU and LFU policies by maintaining two caching lists: one for *recency* and one for *frequency*. ARC dynamically changes the number of cache-lines allocated for each list to either favor recency or frequency of access in response to workload changes. In addition, ARC uses shadow queues to track more keys beyond the cache size. This helps ARC to maintain a broader perspective of the access distribution beyond the cache size. ARC is designed to find the fine balance between recent and frequent accesses. As a result, ARC pays the cost of caching every new cold key in the recency list evicting a hot key from the frequency list. This cost is significant especially when the cache size is much smaller than the key space and the workload is skewed favoring frequency over recency.

LRU-k [39] tracks the last k accesses of each cached key, in addition to a pre-configured *manually* fixed size history that includes the access information of the recently evicted keys from the cache. New keys replace the cached key with the least recently k^{th} access. The evicted key is moved to the history, which is typically implemented using LRU. LRU-k is a suitable strategy to mock perfect caching of periodically stable skewed workloads when its cache and history sizes are perfectly pre-configured for this specific workload. However, due to the lack of LRU-k's dynamic resizing and elasticity of both its cache and history sizes, we choose to introduce CoT that is designed with native resizing and elasticity functionality. This functionality allows CoT to adapt its cache and tracker sizes in response to workload changes.

5 CACHE ON TRACK (COT)

Front-end caches serve two main purposes: *1) decrease the load on the back-end caching layer* and *2) reduce the load-imbalance among the back-end caching servers*. CoT focuses on the latter goal and considers back-end load reduction a complementary side effect. CoT's design philosophy is to track more keys beyond the cache size. This tracking serves as a filter that prevents cold keys from populating the small cache and therefore, only hot keys can populate the cache. In addition, the tracker and the cache are dynamically and adaptively resized to ensure that the load served by the back-end layer follows a load-balance target.

The idea of tracking more keys beyond the cache size has been widely used in replacement policies such as 2Q [29], MQ [45], LRU-k [39, 40], and ARC [36]. Both 2Q and MQ use multiple LRU queues to overcome the weaknesses of LRU of allowing cold keys to replace warmer keys in the cache. All these policies are desgined for fixed memory size environments. However, in a cloud environment where elastic resources can be requested on-demand, a new cache replacement policy is needed to take advantage of this elasticity.

CoT presents a new cache replacement policy that uses a *shadow heap* to track more keys beyond the cache size. Previous works have established the efficiency of heaps in tracking frequent items [37]. In this section, we explain how CoT uses tracking beyond the cache size to achieve the caching objectives listed in Section 4. In particular, CoT answers the following questions: 1) *how to prevent cold keys from replacing hotter keys in the cache?*, 2) *how to reduce the required front-end cache size that achieves lookup load-balance?*, 3) *how to adaptively resize the cache in response to changes in the workload distribution?* and finally 4) *how to dynamically retire old heavy hitters?*.

5.1 Notation

The key space, denoted by S, is assumed to be large in the scale of trillions of keys. Each front-end server maintains a cache of size $C <<< S$. The set of cached keys is denoted by S_c. To capture the top-C hottest keys, each front-end server tracks $K > C$ keys. The set of tracked key is denoted by S_k. Front-end servers cache the top-C hottest keys where $S_c \subset S_k$. A key hotness h_k is determined using the dual cost model introduced in [18]. In this model, read accesses increase a key hotness by a read weight r_w while update accesses decrease it by an update weight u_w. As update accesses cause cache invalidations. Therefore, frequently updated keys should not be cached and thus an update access decreases a key's hotness. For each tracked key, the read count $k.r_c$ and the update count $k.u_c$ are maintained to capture the number of read and update accesses of this key. Equation 1 shows how the hotness of key k is calculated. We use $r_w = u_w = 1$ in our experiments.

$$h_k = k.r_c \times r_w - k.u_c \times u_w \quad (1)$$

h_{min} refers to the minimum key hotness in the cache. h_{min} splits the tracked keys into *two* subsets: 1) the set of cached keys (also tracked) S_c of size C and 2) the set of tracked but not cached

S	key space
K	number of tracked keys at the front-end
C	number of cached keys at the front-end
h_k	hotness of a key k
$k.r_c$	read count of a key k
$k.u_c$	update count of a key k
r_w	the weight of a read operation
u_w	the weight of an update operation
h_{min}	the minimum hotness of keys in the cache
S_k	the set of all tracked keys
S_c	the set of cached keys (these keys are also tracked)
S_{k-c}	the set of tracked but not cached keys
I_c	the current local lookup load-imbalance
I_t	the target lookup load-imbalance
α	the average hit-rate per cache-line in an epoch
E	Epoch: a configurable number of accesses

Table 1: Summary of notation.

keys S_{k-c} of size $K-C$. The current local load-imbalance among caching servers lookup load is denoted by I_c. I_c is a local variable at each front-end that determines the current contribution of this front-end to the back-end load-imbalance. I_c is defined as the workload ratio between the most loaded back-end server and the least loaded back-end server as observed at a front-end server. For example, if a front-end server sends, during an epoch, a maximum of 5K key lookups to some back-end server and, during the same epoch, a minimum of 1K key lookups to another back-end server then I_c, at this front-end, equals 5. I_t is the target load-imbalance among the caching servers. I_t is the only input parameter set by the system administrator and is used by front-end servers to dynamically adjust their cache and tracker sizes. Ideally I_t should be set close to 1. $I_t = 1.1$ means that back-end load-balance is achieved if the most loaded server observes at most 10% more key lookups than the least loaded server. Finally, we define another **local auto-adjusted** parameter α. α is the average hits per cache-line and it determines the quality of the cached keys. A cache-line, or entry, contains one cached key and its corresponding value and α determines the average hits per cache-line during an epoch. For example, if a cache consists of 10 cache-lines and they cumulatively observe 2000 hits in an epoch, we consider α to be 200. α helps detect changes in workload and adjust the cache size accordingly. Note that CoT automatically infers the value of α based on the observed workload. Hence, the system administrator does not need to set the value of α. E is an epoch parameter defined by a configurable number of accesses. CoT runs its dynamic resizing algorithm every E accesses. Table 1 summarizes the notation.

5.2 Space-Saving Tracking Algorithm

CoT uses the *space-saving* algorithm introduced in [37] to track the key hotness at front-end servers. Space-saving uses a min-heap to order keys based on their hotness and a hashmap to lookup keys in the tracker in *O(1)*. The space-saving algorithm is shown in Algorithm 1. If the accessed key k is not in the tracker (Line 1), it replaces the key with minimum hotness at the root of the min-heap (Lines 2, 3, and 4). The algorithm gives the newly added key the benefit of doubt and assigns it the hotness of the replaced key. As a result, the newly added key gets the opportunity to survive immediate replacement in the tracker. Whether the accessed key k was in the tracker or is newly added to the tracker, the hotness of the key is updated based on the access type according to Equation 1 (Line 6) and the heap is accordingly adjusted (Line 7).

Algorithm 1 The space-saving algorithm: track_key(key k, access_type t).

State: S_k: keys in the tracker.
Input: (key k, access_type t)

1: **if** $k \notin S_k$ **then**
2: let k' be the root of the min-heap
3: replace k' with k
4: $h_k := h_{k'}$
5: **end if**
6: h_k := update_hotness(k, t)
7: adjust_heap(k)
8: return h_k

5.3 CoT: Cache Replacement Policy

CoT's tracker captures the approximate top K hot keys. Each front-end server should cache the exact top C keys out of the tracked K keys where $C < K$. The exactness of the top C cached keys is considered with respect to the approximation of the top K tracked keys. Caching the exact top C keys prevents cold and noisy keys from replacing hotter keys in the cache and achieves the first caching objective. To determine the exact top C keys, CoT maintains a cache of size C in a min-heap structure. Cached keys are partially ordered in the min-heap based on their hotness. The root of the cache min-heap gives the minimum hotness, h_{min}, among the cached keys. h_{min} splits the tracked keys into *two unordered* subsets S_c and S_{k-c} such that:

- $|S_c| = C$ and $\forall_{x \in S_c} h_x \geq h_{min}$
- $|S_{k-c}| = K - C$ and $\forall_{x \in S_{k-c}} h_x < h_{min}$

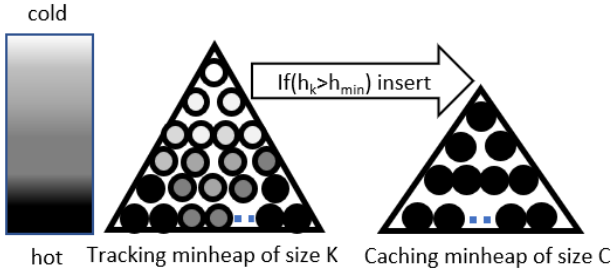


Figure 2: CoT: a key is inserted to the cache if its hotness exceeds the minimum hotness of the cached keys.

For every key access, the hotness information of the accessed key is updated in the tracker. If the accessed key is cached, its hotness information is updated in the cache as well. However, if the accessed key is not cached, its hotness is compared against h_{min}. As shown in Figure 2, the accessed key is inserted into the cache only if its hotness exceeds h_{min}. Algorithm 2 explains the details of CoT's cache replacement algorithm.

For every key access, the *track_key* function of Algorithm 1 is called (Line 1) to update the tracking information and the hotness of the accessed key. Then, a key access is served from the local cache only if the key is in the cache (Lines 3). Otherwise, the access is served from the caching server (Line 5). Serving an access from the local cache implicitly updates the accessed key hotness and location in the cache min-heap. If the accessed key

Algorithm 2 CoT's caching algorithm

State: S_k: keys in the tracker and S_c: keys in the cache.
Input: (key k, access_type t)

```
1:  h_k = track_key(k, t) as in Algorithm 1
2:  if k ∈ S_c then
3:      let v = access(S_c, k) // local cache access
4:  else
5:      let v = server_access(k) // caching server access
6:      if h_k > h_min then
7:          insert(S_c, k, v) // local cache insert
8:      end if
9:  end if
10: return v
```

is not cached, its hotness is compared against h_{min} (Line 6). The accessed key is inserted to the local cache if its hotness exceeds h_{min} (Line 7). This happens only if there is a tracked but not cached key that is hotter than one of the cached keys. Keys are inserted to the cache together with their tracked hotness information. Inserting keys into the cache follows the LFU replacement policy. This implies that a local cache insert (Line 7) would result in the replacement of the coldest key in the cache (the root of the cache heap) if the local cache is full.

5.4 CoT: Adaptive Cache Resizing

This section explains CoT's resizing algorithm. This algorithm reduces the necessary front-end cache size that achieves back-end lookup load-balance. In addition, this algorithm dynamically expands or shrinks CoT's tracker and cache sizes when the served workload changes. Also, this algorithm detects changes in the set of hot keys and retires old hot keys that are not hot any more. As explained in Section 1, reducing the front-end cache size decreases the front-end cache monetary cost, limits the overheads of data consistency management techniques, and maximizes the benefit of front-end caches measured by the average cache-hits per cache-line and back-end load-imbalance reduction.

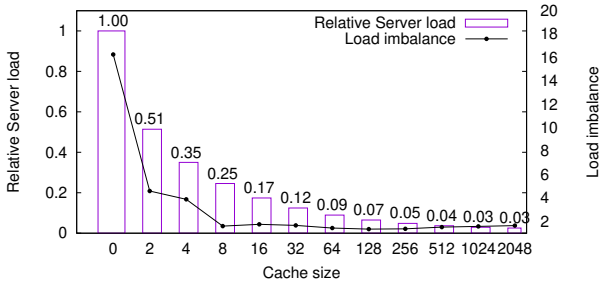


Figure 3: Reduction in relative server load and load-imbalance among caching servers as front-end cache size increases.

The Need for Cache Resizing: Figure 3 experimentally shows the effect of increasing the front-end cache size on both back-end *load-imbalance reduction* and decreasing the workload at the back-end. In this experiment, 8 memcached shards are deployed to serve back-end lookups and 20 clients send lookup requests following a significantly skewed Zipfian distribution (s = 1.5). The size of the key space is 1 million and the total number of lookups is 10 millions. The front-end cache size at each client is varied from 0 cachelines (no cache) to 2048 cachelines (≈0.2% of the key space). Front-end caches use CoT's replacement policy and a ratio of 4:1 is maintained between CoT's tracker size and CoT's cache size. We define back-end load-imbalance as the workload ratio between the most loaded server and the least loaded server. The target load-imbalance I_t is set to 1.5. As shown in Figure 3, processing all the lookups from the back-end caching servers (front-end cache size = 0) leads to a significant load-imbalance of 16.26 among the caching servers. This means that the most loaded caching server receives 16.26 times the number of lookup requests received by the least loaded caching server. As the front-end cache size increases, the server size load-imbalance drastically decreases. As shown, a front-end cache of size 64 cache lines at each client reduces the load-imbalance to 1.44 (an order of magnitude less load-imbalance across the caching servers) achieving the target load-imbalance I_t = 1.5. Increasing the front-end cache size beyond 64 cache lines only reduces the back-end aggregated load but not the back-end load-imbalance. The *relative server load* is calculated by comparing the server load for a given front-end cache size to the server load when there is no front-end caching (cache size = 0). Figure 3 demonstrates the reduction in the relative server load as the front-end cache size increases. However, the benefit of doubling the cache size proportionally decays with the key hotness distribution. As shown in Figure 3, the first 64 cachelines reduce the relative server load by 91% while the second 64 cachelines reduce the relative server load by only 2% more.

The failure of the "one size fits all" design strategy suggests that statically allocating fixed cache and tracker sizes to all front-end servers is not ideal. Each front-end server should independently and adaptively be configured according to the key access distribution it serves. Also, changes in workloads can alter the key access distribution, the skew level, or the set of hot keys. For example, social networks and web front-end servers that serve different geographical regions might experience different key access distributions and different local trends (e.g., #miami vs. #ny). Therefore, CoT's cache resizing algorithm learns the key access distribution independently at each front-end and dynamically resizes the cache and the tracker to achieve lookup load-imbalance target I_t. CoT is designed to reduce the front-end cache size that achieves I_t. Any increase in the front-end cache size beyond CoT's recommendation mainly decreases back-end load and should consider other conflicting parameters such as the additional cost of the memory cost, the cost of updates and maintaining the additional cached keys, and the percentage of back-end load reduction that results from allocating additional front-end caches.

Cache Resizing Algorithm (parameter configuration): Front-end servers use CoT to minimize the cache size that achieves a target load-imbalance I_t. Initially, front-end servers are configured with no front-end caches. The system administrator configures CoT by an input *target load-imbalance* parameter I_t that determines the maximum tolerable imbalance between the most loaded and least loaded back-end caching servers. Afterwards, CoT expands both tracker and cache sizes until the current load-imbalance achieves the inequality $I_c \leq I_t$.

Algorithm 3 describes CoT's cache resizing algorithm. CoT divides the timeline into epochs and each epoch consists of E accesses. Algorithm 3 is executed at the end of each epoch. The epoch size E is proportional to the tracker size K and is dynamically updated to guarantee that $E \geq K$ (Line 3). This condition helps ensure that CoT does not trigger consecutive resizes before the cache and the tracker are warmed up with keys. During each epoch, CoT tracks the number of lookups sent to every back-end caching server. In addition, CoT tracks the total number of cache

hits and tracker hits during this epoch. At the end of each epoch, CoT calculates the current load-imbalance I_c as the ratio between the highest and the lowest load on back-end servers during this epoch. Also, CoT calculates the current average hit per cached key α_c. α_c equals the total cache hits in the current epoch divided by the cache size. Similarly, CoT calculates the current average hit per tracked but not cache key α_{k-c}. CoT compares I_c to I_t and decides on a resizing action as follows.

(1) $I_c > I_t$ and $\alpha_c \geq \alpha_{k-c}$ (Line 1), this means that the target load-imbalance is not achieved and cached keys observe more hits than the keys in the tracker. CoT follows the binary search algorithm in searching for the front-end cache size that achieves I_t. Therefore, CoT decides to double the front-end cache size (Line 2). As a result, CoT doubles the tracker size as well to maintain a tracker to cache size ratio of at least 2, $K \geq 2 \cdot C$ (Line 2). The cache and tracker sizes are doubled until either I_t is achieved or a configurable upper limit cache size is hit. In addition, CoT uses a local variable α_t to capture the quality of the cached keys when I_t is first achieved. Initially, $\alpha_t = 0$. CoT then sets α_t to the average hits per cache-line α_c during the current epoch (Line 4). In subsequent epochs, α_t is used to detect changes in workload.

(2) $I_c \leq I_t$ (Line 5), this means that the target load-imbalance has been achieved. However, changes in workload could alter the quality of the cached keys. Therefore, CoT uses α_t to detect and handle changes in workload in future epochs as explained below.

Algorithm 3 CoT's elastic resizing algorithm.

State: S_c: keys in the cache, S_k: keys in the tracker, C: cache capacity, K: tracker capacity, α_c: average hits per key in S_c in the current epoch, α_{k-c}: average hits per key in S_{k-c} in the current epoch, I_c: current load-imbalance, and α_t: target average hit per key

Input: I_t

```
1:  if I_c > I_t && α_c >= α_{k-c} then
2:      resize(S_c, 2 × C) , resize(S_k, 2 × K)
3:      E := max (E, K)
4:      Let α_t = α_c
5:  else
6:      if α_c < (1 − ε).α_t and α_{k-c} < (1 − ε).α_t then
7:          resize(S_c, C/2), resize(S_k, K/2)
8:      else if α_c < (1 − ε).α_t and α_{k-c} > (1 − ε).α_t then
9:          half_life_time_decay()
10:     end if
11: end if
```

α_t is reset whenever the inequality $I_c \leq I_t$ is violated and Algorithm 3 expands cache and tracker sizes. Ideally, when the inequality $I_c \leq I_t$ holds, keys in the cache (the set S_c) achieve α_t hits per cache-line during every epoch while keys in the tracker but not in the cache (the set S_{k-c}) do not achieve α_t. This happens because keys in the set S_{k-c} are colder than keys in the set S_c. α_t represents a target hit-rate per cache-line for future epochs. Therefore, if keys in the cache do not meet the target α_t in a following epoch, this indicates that the quality of the cached keys has changed and an action needs to be taken as follows.

(1) Case 1: keys in S_c, on the average, do not achieve α_t hits per cacheline and keys in S_{k-c} do not achieve α_t hits as well (Line 6). This indicates that the quality of the cached keys decreased. In response. CoT shrinks both the cache and the tracker sizes (Line 7). If shrinking both cache and tracker sizes results in a violation of the inequality $I_c < I_t$, Algorithm 3 doubles both tracker and cache sizes in the following epoch and α_t is reset as a result. In Line 6, we compare the average hits per key in both S_c and S_{k-c} to $(1 - \epsilon) \cdot \alpha_t$ instead of α_t. Note that ϵ is a small constant $<<<$ 1 that is used to avoid unnecessary resizing actions due to insignificant statistical variations.

(2) Case 2: keys in S_c do not achieve α_t while keys in S_{k-c} achieve α_t (Line 8). This signals that the set of hot keys is changing and keys in S_{k-c} are becoming hotter than keys in S_c. For this, CoT triggers a half-life time decaying algorithm that halves the hotness of all cached and tracked keys (Line 9). This decaying algorithm aims to forget old trends that are no longer hot to be cached (e.g., Gangnam style song). Different decaying algorithms have been developed in the literature [14, 16, 17]. Therefore, this paper only focuses on the resizing algorithm details without implementing a specific decaying algorithm.

(3) Case 3: keys in S_c achieve α_t while keys in S_{k-c} do not achieve α_t. This means that the quality of the cached keys has not changed and therefore, CoT does not take any action. Similarly, if keys in both sets S_c and S_{k-c} achieve α_t, CoT does not take any action as long as the inequality $I_c < I_t$ holds.

6 EXPERIMENTAL EVALUATION

This section evaluates both CoT's caching and adaptive resizing algorithms. We choose to compare CoT to traditional and widely used replacement policies like LRU and LFU. In addition, we compare CoT to both ARC [36] and LRU-k [39]. As stated in [36], ARC, in its online auto-configuration setting, achieves comparable performance to LRU-2 (which is the most responsive LRU-k) [39, 40], 2Q [29], LRFU [31], and LIRS [27] even when these policies are perfectly tuned offline. Also, ARC outperforms the online adaptive replacement policy MQ [45]. Therefore, we compare with ARC and LRU-2 as representatives of these different polices. Section 6.1 explains the experimental setup. First, we compare the hit rates of CoT's cache algorithm to LRU, LFU, ARC, and LRU-2 hit rates for different front-end cache sizes in Section 6.2. Then, we compare the required front-end cache size for each replacement policy to achieve a target back-end load-imbalance I_t in Section 6.3. In Section 6.4, we provide an end-to-end evaluation of front-end caches comparing the end-to-end performance of CoT, LRU, LFU, ARC, and LRU-2 on different workloads with the configuration where no front-end cache is deployed. Finally, CoT's resizing algorithm is evaluated in Section 6.5.

6.1 Experiment Setup

We deploy 8 instances of memcached [3] on a small cluster of 4 caching servers (2 memcached instance per server). Each caching server has an Intel(R) Xeon(R) CPU E3-1235 (8MB Cache and 16GB RAM) with 4GB RAM dedicated to each memcached instance. Caching servers and clients run ubuntu 18.04 and connected to the same 10Gbps Ethernet network. No explicit network or OS optimization are used.

A dedicated client machine with Intel(R) Core(TM) i7-6700HQ CPU and 16GB of RAM is used to generate client workloads. The client machine executes multiple client threads to submit workloads to caching servers. Client threads use Spymemcached

2.11.4 [5], a Java-based memcached client, to communicate with memcached cluster. Spymemcached provides communication abstractions that distribute workload among caching servers using *consistent hashing* [30]. We slightly modified Spymemcached to monitor the workload per back-end server at each front-end. Client threads use Yahoo! Cloud Serving Benchmark (YCSB) [15] to generate workloads for the experiments. YCSB is a standard key/value store benchmarking framework. YCSB is used to generate key/value store requests such as *Get*, *Set*, and *Insert*. YCSB enables configuring the ratio between read (Get) and write (Set) accesses. Also, YCSB allows the generation of accesses that follow different access distributions. As YCSB is CPU-intensive, the client machine runs at most 20 client threads per machine to avoid contention among client threads. During our experiments, we realized that YCSB's ScrambledZipfian workload generator has a bug as it generates Zipfian workload distributions with significantly less skew than the skew level it is configured with. Therefore, we use YCSB's Zipfian generator instead of YCSB's ScrambledZipfian. Figure 4 shows the hits per key generated for the top 1024 out 1 million keys and 100 million samples of a zipfian 0.99 distribution from YCSB-Zipfian, YCSB-ScambledZipfian, and theoretical zipfian generators. As shown, YCSB-Zipfian generator (CDF 0.512) is almost identical to the theoretical generator (CDF 0.504) while YCSB-ScrambledZipfian generator (CDF 0.30) has much less skew than the theoretical zipfian generator (40% less hits in the top 1024 keys than the theoretical distribution).

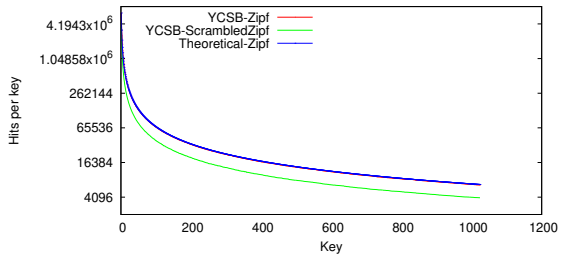


Figure 4: YCSB-Zipf vs ScrambledZipf vs Theoretical-Zipf

Our experiments use different variations of YCSB core workloads. Workloads consist of 1 million key/value pairs. Each key consists of a common prefix *"usertable:"* and a unique ID. We use a value size of 750 KB making a dataset of size 715GB. Experiments use read intensive workloads that follow Tao's [9] read-to-write ratio of 99.8% reads and 0.2% updates. Unless otherwise specified, experiments consist of 10 million key accesses sampled from different access distributions such as Zipfian (s = 0.90, 0.99, or 1.2) and uniform. Client threads submit access requests back-to-back. Each client thread can have only one outgoing request. Clients submit a new request as soon as they receive an acknowledgement for their outgoing request.

6.2 Hit Rate

The first experiment compares CoT's hit rate to LRU, LFU, ARC, and LRU-2 hit rates using equal cache sizes for all replacement policies. 20 client threads are provisioned on one client machine and each cache client thread maintains its own cache. **Configuration:** The cache size is varied from a very small cache of 2 cache-lines to 2048 cache-lines. The hit rate is compared using different Zipfian access distributions with skew parameter values s = 0.90, 0.99, and 1.2 as shown in Figures 5(a), 5(b), and 5(c) respectively. In addition, 14 days of Wikipedia's real traces, collected by [42], are used to compare CoT to other replacement policies, including LHD [8], as shown in Figure 5(d). CoT's tracker to cache size ratio determines how many tracking nodes are used for every cache-line. CoT automatically detects the ideal tracker to cache ratio for any workload by fixing the cache size and doubling the tracker size until the observed hit-rate gains from increasing the tracker size are insignificant i.e., the observed hit-rate saturates. The tracker to cache size ratio decreases as the workload skew increases. A workload with high skew simplifies the task of distinguishing hot keys from cold keys and hence, CoT requires a smaller tracker size to successfully filter hot keys from cold keys. Note that LRU-2 is also configured with the same history to cache size as CoT's tracker to cache size. In this experiment, for each skew level, CoT's tracker to cache size ratio is varied as follows: 16:1 for Zipfian 0.9 and Wikipedia, 8:1 for Zipfian 0.99, and 4:1 for Zipfian 1.2. Note that CoT's tracker maintains only the meta-data of tracked keys. Each tracker node consists of a read counter and a write counter with 8 bytes of memory overhead per tracking node. In real-world workloads, value sizes vary from tens of KBs to few MBs. For example, Google's Bigtable [10] uses a value size of 64 KB. Therefore, a memory overhead of at most $\frac{1}{8}$ KB (16 tracker nodes * 8 bytes) per cache-line is negligible (0.2%).

In Figures 5, the *x-axis* represents the cache size expressed as the number of cache-lines. The *y-axis* represents the front-end cache hit rate (%) as a percentage of the total workload size. At each cache size, the cache hit rates are reported for LRU, LFU, ARC, LRU-2, and CoT cache replacement policies. In addition, TPC represents the theoretically calculated hit-rate from the Zipfian distribution CDF if a perfect cache with the same cache size is deployed. For each experiment, the TPC is configured with the same key space size N = (1 million keys), the same sample generated size (10 million samples), the same skew parameter s (e.g., 0.9, 0.99, 1.2) of the experiment, and k equals to the cache size (ranging from 2 – 2048). For example, a perfect cache of size 2 cache-lines stores the top-2 hot keys and hence any access to these 2 keys results in a cache hit while accesses to other keys result in cache misses. For Wikipedia traces, the TPC is the cumulative accesses of the top-C keys.

Analysis: As shown in Figure 5(a), CoT surpasses LRU, LFU, ARC, and LRU-2 hit rates at all cache sizes. In fact, CoT achieves almost similar hit-rate to the TPC hit-rate. In Figure 5(a), CoT outperforms TPC for some cache size which is counter intuitive. This happens as TPC is theoretically calculated using the Zipfian CDF while CoT's hit-rate is calculated out of YCSB's sampled distributions which are approximate distributions. In addition, CoT achieves higher hit-rates than both LRU and LFU with **75% less cache-lines**. As shown, CoT with 512 cache-lines achieves 10% more hits than both LRU and LFU with 2048 cache-lines. Also, CoT achieves higher hit rate than ARC using **50% less cache-lines**. In fact, CoT configured with 512 cache-lines achieves 2% more hits than ARC with 1024 cache-lines. Taking tracking memory overhead into account, CoT maintains a tracker to cache size ratio of 16:1 for this workload (Zipfian 0.9). This means that CoT adds an overhead of 128 bytes (16 tracking nodes * 8 bytes each) per cache-line. The percentage of CoT's tracking memory overhead decreases as the cache-line size increases. For example, CoT introduces a tracking overhead of 0.02% when the cache-line size is 750KB. Finally, CoT consistently achieves 8-10% higher hit-rate than LRU-2 configured with the same history and cache sizes as CoT's tracker and cache sizes.

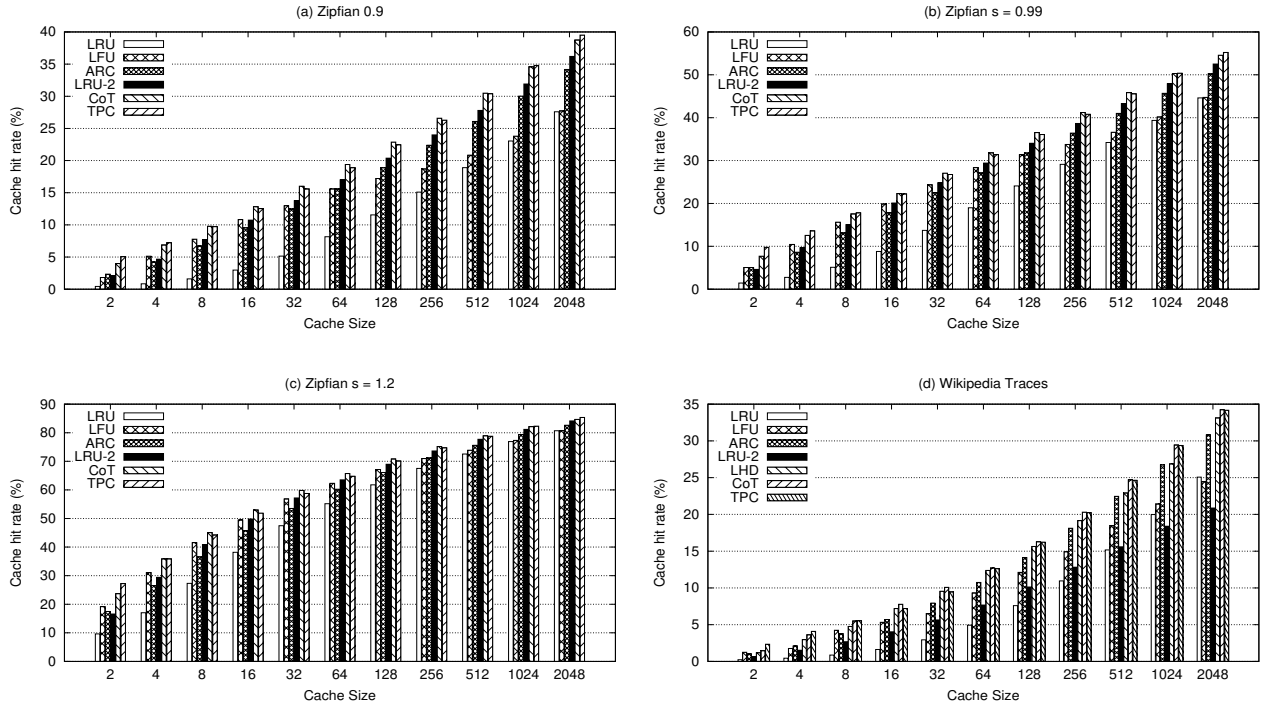


Figure 5: Comparison of LRU, LFU, ARC, LRU-2, LHD, CoT and TPC's hit rates using various Zipfian and Wikipedia traces

Similarly, as illustrated in Figures 5(b) and 5(c), CoT outpaces LRU, LFU, ARC, and LRU-2 hit rates at all different cache sizes. Figure 5(b) shows that a configuration of CoT using 512 cache-lines achieves 3% more hits than both configurations of LRU and LFU with 2048 cache-lines. Also, CoT consistently outperforms ARC's hit rate with 50% less cache-lines. Finally, CoT achieves 3-7% higher hit-rate than LRU-2 configured with the same history and cache sizes. Figures 5(b) and 5(c) highlight that increasing workload skew decreases the advantage of CoT. As workload skew increases, the ability of LRU, LFU, ARC, LRU-2 to distinguish between hot and cold keys increases and hence CoT's preeminence decreases.

In addition to YCSB synthetic traces, Wikipedia real traces show in Figure 5(d) that CoT significantly outperforms other replacement policies at every cache size. This Wikipedia real traces comparison includes LHD [8], a recently proposed caching policy specifically for cloud applications. As Figure 5(d) shows, CoT is the only replacement policy that achieves almost the same hit rate of the cumulative top-C keys beating LRU, LFU, ARC, LRU-2, and LHD by 23-84%, 14-51%, 9-42%, 37-58%, and 4-24% respectively for different cache sizes.

6.3 Back-End Load-Imbalance

In this section, we compare the required front-end cache sizes for different replacement policies to achieve a back-end load-imbalance target I_t. **Configuration:** Different skewed workloads are used, namely, Zipfian s = 0.9, s = 0.99, and s = 1.2. For each distribution, we first measure the back-end load-imbalance when no front-end cache is used. A back-end load-imbalance target I_t is set to $I_t = 1.1$. This means that the back-end is load balanced if the most loaded back-end server processes at most 10% more lookups than the least loaded back-end server. We evaluate the back-end load-imbalance while increasing the front-end cache size using different cache replacement policies, namely, LRU, LFU, ARC, LRU-2, and CoT. In this experiment, CoT uses the same tracker-to-cache size ratio as in Section 6.2. For each replacement policy, we report the minimum required number of cache-lines to achieve I_t.

Dist.	Load-imbalance	Number of cache-lines to achieve $I_t = 1.1$				
	No cache	LRU	LFU	ARC	LRU-2	CoT
Zipf 0.9	1.35	64	16	16	8	8
Zipf 0.99	1.73	128	16	16	16	8
Zipf 1.20	4.18	2048	2048	1024	1024	512

Table 2: The minimum required number of cache-lines for different replacement policies to achieve a back-end load-imbalance target $I_t = 1.1$ for different workload distributions.

Analysis: Table 2 summarizes the reported results for different distributions using LRU, LFU, ARC, LRU-2, and CoT replacement policies. For each distribution, the initial back-end load-imbalance is measured using no front-end cache. As shown, the initial load-imbalances for Zipf 0.9, Zipf 0.99, and Zipf 1.20 are 1.35, 1.73, and 4.18 respectively. For each distribution, the minimum required number of cache-lines for LRU, LFU, ARC, and CoT to achieve a target load-imbalance of $I_t = 1.1$ is reported. As shown, CoT requires **50% to 93.75% less cache-lines** than other replacement policies to achieve I_t. Since LRU-2 is configured with a history size equals to CoT's tracker size, LRU-2 requires the second least number of cache-lines to achieve I_t.

6.4 End-to-End Evaluation

In this section, we evaluate the effect of front-end caches using LRU, LFU, ARC, LRU-2, and CoT replacement policies on

the overall running time of different workloads. This experiment also demonstrates the overhead of front-end caches on the overall running time. **Configuration:** This experiment uses 3 different workload distributions, namely, uniform, Zipfian (s = 0.99), and Zipfian (s = 1.2) distributions as shown in Figure 6. For all the three workloads, each replacement policy is configured with 512 cache-lines. Also, CoT and LRU-2 maintain a tracker (history) to cache size ratio of 8:1 for Zipfian 0.99 and 4:1 for both Zipfian 1.2 and uniform distributions. In this experiment, a total of 1M accesses are sent to the caching servers by 20 client threads running on one client machine. Each experiment is executed 10 times and the average overall running time with 95% confidence intervals are reported.

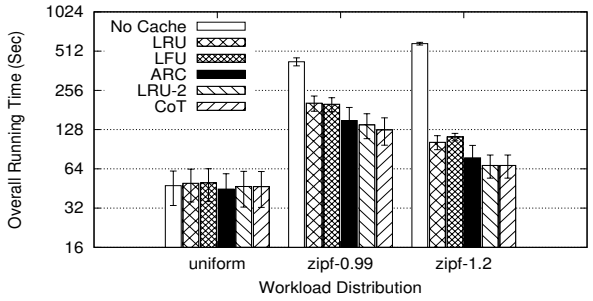


Figure 6: Front-end caching effect on the end-to-end overall running time using different workload distributions.

In this experiment, the front-end servers are allocated in the same cluster as the back-end servers. The average Round-Trip Time (RTT) between front-end machines and back-end machines is 244μs. This small RTT allows us to fairly measure the overhead of front-end caches by minimizing the performance advantages achieved by front-end cache hits. In real-world deployments where front-end servers are deployed in edge-datacenters and the RTT between front-end servers and back-end servers is in order of milliseconds, front-end caches achieve more significant performance gains.

Analysis: The uniform workload is used to measure the overhead of front-end caches. In a uniform workload, all keys in the key space are equally hot and front-end caches cannot take any advantage of workload skew to benefit some keys over others. Therefore, front-end caches only introduce the overhead of maintaining the cache without achieving any significant performance gains. As shown in Figure 6, there is no significant statistical difference between the overall run time when there is no front-end cache and when there is a small front-end cache with different replacement policies. Adding a small front-end cache does not incur run time overhead even for replacement policies that use a heap, e.g., LFU, LRU-2, and CoT.

The workloads Zipfian 0.99 and Zipfian 1.2 are used to show the advantage of front-end caches even when the network delays between front-end servers and back-end servers are minimal. As shown in Figure 6, workload skew results in significant overall running time overhead in the absence of front-end caches. This happens because the most loaded server introduces a performance bottleneck especially under thrashing (managing 20 connections, one from each client thread). As the load-imbalance increases, the effect of this bottleneck is worsen. Specifically, in Figure 6, the overall running time of Zipfian 0.99 and Zipfian 1.2 workloads are respectively 8.9x and 12.27x of the uniform workload when no front-end cache is deployed. Deploying a small front-end cache of 512 cachelines significantly reduces the effect of back-end bottlenecks. Deploying a CoT small cache in the front-end results in 70% running time reduction for Zipfian 0.99 and 88% running time reduction for Zipfian 1.2 in comparison to having no front-end cache. Other replacement policies achieve running time reductions of 52% to 67% for Zipfian 0.99 and 80% to 88% for Zipfian 1.2. LRU-2 achieves the second best average overall running time after CoT with no significant statistical difference between the two policies. Since both policies use the same tracker (history) size, this again suggests that having a bigger tracker helps separate cold and noisy keys from hot keys. Since the ideal tracker to cache size ratio differs from one workload to another, having an automatic and dynamic way to configure this ratio at run-time while serving workload gives CoT a big leap over statically configured replacement policies.

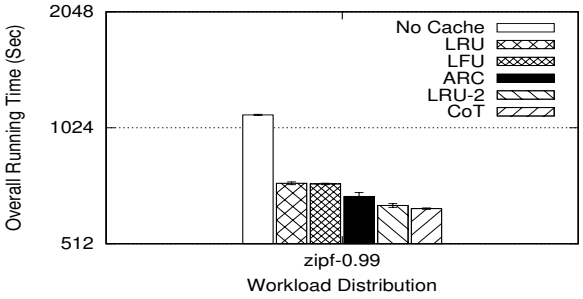


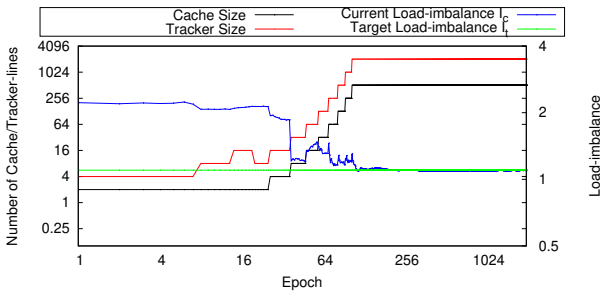
Figure 7: Front-end caching effect on the end-to-end overall running time in Amazon EC2 cloud.

Similar results have been observed in a cloud setup as well. We run the same end-to-end evaluation on Amazon EC2's cloud. 20 client threads run on a t2.xlarge machine in California to send 1M requests following zipfian 0.99 distribution to 8 m3.medium ElastiCache cluster in Oregon. Front-end cache and tracker sizes are similar to the local experiments configuration. The key space consists of 1M keys and each key has a value of 10KB (smaller values are used since network latency is large in comparison to local experiments). As shown in Figure 7, a small front-end cache of 512 cache-lines has significant performance gains in comparison to the setup where no front-end cache is deployed. Also, both CoT and LRU-2 outperform other front-end cache replacement policies. Also, CoT slightly outperforms LRU-2 achieving 2% performance gain on the average. The main advantage of CoT over LRU-2 is its ability to dynamically discover the ideal cache and tracker sizes that achieve backend load-balance as the workload distribution changes. The following section illustrates CoT's performance in response to workload changes.

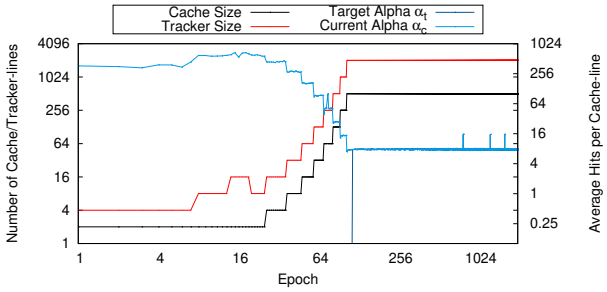
6.5 Adaptive Resizing

This section evaluates CoT's auto-configure and resizing algorithms. **Configuration:** *First*, we configure a front-end client that serves a Zipfian 1.2 workload with a tiny cache of size *two* cachelines and a tracker of size of *four* tracking entries. This experiment aims to show how CoT expands cache and tracker sizes to achieve a target load-imbalance I_t as shown in Figure 8. After CoT reaches the cache size that achieves I_t, the average hit per cache-line α_t is recorded as explained in Algorithm 3. *Second*, we alter the workload distribution to uniform and monitors how CoT shrinks tracker and cache sizes in response to workload changes without violating the load-imbalance target I_t in Figure 9. In both experiments, I_t is set to 1.1 and the epoch

size is 5000 accesses. In both Figures 8a and 9a, the x-axis represents the epoch number, the left y-axis represents the number of tracker and cache lines, and the right y-axis represents the load-imbalance. The black and red lines represent cache and tracker sizes respectively with respect to the left y-axis. The blue and green lines represent the current load-imbalance and the target load-imbalance respectively with respect to the right y-axis. Same axis description applies for both Figures 8b and 9b except that the right y-axis represents the average hit per cache-line during each epoch. Also, the light blue and the dark blue lines represent the current average hit per cache-line and the target hit per cache-line at each epoch with respect to the right y-axis.



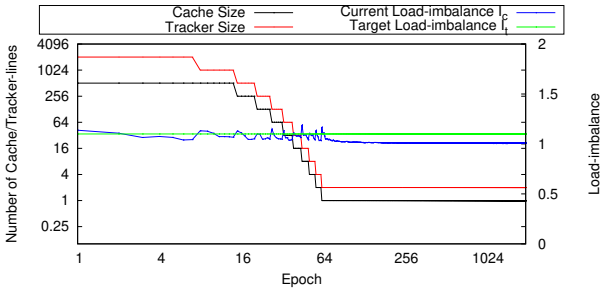
(a) Changes in cache and tracker sizes and the current load-imbalance I_c over epochs.



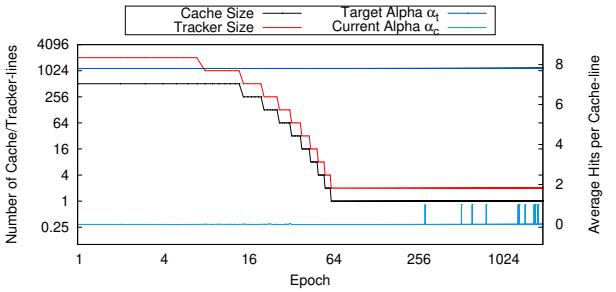
(b) Changes in cache and tracker sizes and the current hit rate per cacheline α_c over epochs.

Figure 8: CoT adaptively expands tracker and cache sizes to achieve a target load-imbalance $I_t = 1.1$ for a Zipfian 1.2 workload.

Analysis: In Figure 8a, CoT is initially configured with a cache of size 2 and a tracker of size 4. CoT's resizing algorithm runs in 2 phases. In the first phase, CoT discovers the ideal tracker-to-cache size ratio that maximizes the hit rate for a fixed cache size for the current workload. For this, CoT fixes the cache size and doubles the tracker size until doubling the tracker size achieves no significant benefit on the hit rate. This is shown in Figure 8b in the first 15 epochs. CoT allows a warm up period of 5 epochs after each tracker or cache resizing decision. Notice that increasing the tracker size while fixing the cache size reduces the current load-imbalance I_c (shown in Figure 8a) and increases the current observed hit per cache-line α_c (shown in Figure 8b). Figure 8b shows that CoT first expands the tracker size to 16 and during the warm up epochs (epochs 10-15), CoT observes no significant benefit in terms of α_c when compared to a tracker size of 8. In response, CoT therefore shrinks the tracker size to 8 as shown in the dip in the red line in Figure 8b at epoch 16. Afterwards, CoT starts phase 2 searching for the smallest cache size that achieves I_t. For this, CoT doubles the tracker and caches sizes until the target load-imbalance is achieved and the inequality $I_c \leq I_t$ holds as shown in Figure 8a. CoT captures α_t when I_t is first achieved. α_t determines the quality of the cached keys when I_t is reached for the first time. In this experiment, CoT does not trigger resizing if I_c is within 2% of I_t. Also, as the cache size increases, α_c decreases as the skew of the additionally cached keys decreases. For a Zipfian 1.2 workload and to achieve $I_t = 1.1$, CoT requires 512 cache-lines and 2048 tracker lines and achieves an average hit per cache-line of $\alpha_t = 7.8$ per epoch.



(a) Changes in cache and tracker sizes and the current load-imbalance I_c over epochs.



(b) Changes in cache and tracker sizes and the current hit rate per cache-line α_c over epochs.

Figure 9: CoT adaptively shrinks tracker and cache sizes in response to changing the workload to uniform.

Figure 9 shows how CoT successfully shrinks tracker and cache sizes in response to workload skew drop without violating I_t. After running the experiment in Figure 8, we alter the workload to uniform. Therefore, CoT detects a drop in the current average hit per cache-line as shown in Figure 9b. At the same time, CoT observe that the current load-imbalance I_c achieves the inequality $I_c \leq I_t = 1.1$. Therefore, CoT decides to shrink both the tracker and cache sizes until either $\alpha_c \approx \alpha_t = 7.8$ or I_t is violated or until cache and tracker sizes are negligible. First, CoT resets the tracker to cache size ratio to 2:1 and then searches for the right tracker to cache size ratio for the current workload. Since the workload is uniform, expanding the tracker size beyond double the cache size achieves no hit-rate gains as shown in Figure 9b. Therefore, CoT moves to the second phase of shrinking both tracker and cache sizes as long α_t is not achieved and I_t is not violated. As shown, in Figure 9, CoT shrinks both the tracker and the cache sizes until front-end cache size becomes negligible. As shown in Figure 9a, CoT shrinks cache and tracker sizes while ensuring that the target load-imbalance is not violated.

7 CONCLUSION

Cache on Track (CoT) is a decentralized, elastic and predictive cache at the edge of a distributed cloud-based caching infrastructure. CoT's novel cache replacement policy is specifically tailored

for small front-end caches that serve skewed workloads. Using CoT, system administrators do not need to statically specify the cache size at each front-end. Instead, they specify a target back-end load-imbalance I_t and CoT dynamically adjusts front-end cache sizes to achieve I_t. Our experiments show that CoT's replacement policy outperforms the hit-rates of LRU, LFU, ARC, and LRU-2 for the same cache size on different skewed workloads. CoT achieves a target server size load-imbalance with 50% to 93.75% less front-end cache in comparison to other replacement policies. Finally, our experiments show that CoT successfully auto-configures the size of front-end caches in the presence of workload distribution changes.

8 ACKNOWLEDGEMENT

We would like to thank Prabal Saha for his help setting up the experiments on EC2 instances. This work is funded by NSF grants CNS-1703560 and CNS-1815733.

REFERENCES

[1] 2018. Amazon ElastiCache in-memory data store and cache. https://aws.amazon.com/elasticache/.
[2] 2018. Azure Redis Cache. https://azure.microsoft.com/en-us/services/cache/.
[3] 2018. Memcached. A distributed memory object caching system. https://memcached.org/.
[4] 2018. Redis. http://redis.io/.
[5] 2018. A simple, asynchronous, single-threaded memcached client written in java. http://code.google.com/p/spymemcached/.
[6] Atul Adya, John Dunagan, and Alec Wolman. 2010. Centrifuge: Integrated lease management and partitioning for cloud services. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. USENIX Association, 1–1.
[7] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer: Auto-Sharding for Datacenter Applications.. In *OSDI*. 739–753.
[8] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 389–403. https://www.usenix.org/conference/nsdi18/presentation/beckmann
[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.
[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.
[11] Yue Cheng, Aayush Gupta, and Ali R Butt. 2015. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 4.
[12] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*.
[13] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches.. In *NSDI*. 379–392.
[14] Edith Cohen and Martin Strauss. 2003. Maintaining time-decaying stream aggregates. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 223–233.
[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.
[16] Graham Cormode, Flip Korn, and Srikanta Tirthapura. 2008. Exponentially decayed aggregates on data streams. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*. IEEE, 1379–1381.
[17] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. 2009. Forward decay: A practical time decay model for streaming systems. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 138–149.
[18] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. 2017. Caching with Dual Costs. In *Proceedings of the 26th International Conference on World Wide Web Companion*. International World Wide Web Conferences Steering Committee, 643–652.
[19] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 371–384.
[20] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2011. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 23.
[21] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. 2018. Scale-out ccNUMA: exploiting skew with strongly consistent caching. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 21.
[22] Shahram Ghandeharizadeh, Marwan Almaymoni, and Haoyu Huang. 2019. Rejig: a scalable online algorithm for cache server configuration changes. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLII*. Springer, 111–134.
[23] Shahram Ghandeharizadeh and Hieu Nguyen. 2019. Design, implementation, and evaluation of write-back policy with cache augmented data stores. *Proceedings of the VLDB Endowment* 12, 8 (2019), 836–849.
[24] Yu-Ju Hong and Mithuna Thottethodi. 2013. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 13.
[25] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. 2014. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 8.
[26] Jinho Hwang and Timothy Wood. 2013. Adaptive Performance-Aware Distributed Memory Caching.. In *ICAC*, Vol. 13. 33–43.
[27] Song Jiang and Xiaodong Zhang. 2002. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review* 30, 1 (2002), 31–42.
[28] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 121–136.
[29] Theodore Johnson and Dennis Shasha. 1994. X3: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th VLDB Conference*.
[30] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 654–663.
[31] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers* 50, 12 (2001), 1352–1361.
[32] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: a holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 429–444.
[33] David Lomet. 2018. Caching Data Stores: High Performance at Low Cost. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1661–1661.
[34] David Lomet. 2018. Cost/performance in modern data stores: how data caching systems succeed. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*. ACM, 9.
[35] David B Lomet. 2019. Data Caching Systems Win the Cost/Performance Game. *IEEE Data Eng. Bull.* 42, 1 (2019), 3–5.
[36] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache.. In *FAST*, Vol. 3. 115–130.
[37] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*. Springer, 398–412.
[38] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 385–398.
[39] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.
[40] Elizabeth J O'neil, Patrick E O'Neil, and Gerhard Weikum. 1999. An optimality proof of the LRU-K page replacement algorithm. *Journal of the ACM (JACM)* 46, 1 (1999), 92–112.
[41] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.
[42] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. 2020. Learning relaxed belady for content distribution network caching. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 529–544.
[43] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. 2019. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment* 12, 6 (2019), 624–638.
[44] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. 2017. Caching at the web scale. *Proceedings of the VLDB Endowment* 10, 12 (2017), 2002–2005.
[45] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches.. In *USENIX Annual Technical Conference, General Track*. 91–104.

Knowledge Graph Management on the Edge

Weiqin Xu
LIGM Univ Paris Est Marne la Vallée, CNRS, France
Engie Lab, CRIGEN, Stains, France
weiqin.xu@u-pem.fr

Olivier Curé
LIGM Univ Paris Est Marne la Vallée, CNRS, France
olivier.cure@u-pem.fr

Philippe Calvez
Engie Lab, CRIGEN
Stains, France
philippe.calvez1@engie.com

ABSTRACT

Edge computing emerges as an innovative platform for services requiring low latency decision making. Its success partly depends on the existence of efficient data management systems. We consider that knowledge graph management systems have a key role to play in this context due to their data integration and reasoning features. In this paper, we present SuccinctEdge that can answer SPARQL queries, including those requiring reasoning services associated to some ontology. We provide details on its design and implementation before demonstrating its efficiency on real-world and synthetic data sets.

1 INTRODUCTION

Edge computing[2] corresponds to a processing paradigm that brings storage, management, and processing of huge amounts of data closer to the location where it needs to be performed. As such, this emerging trend complements a cloud computing approach by supporting the design of highly local context aware and responsive services, hence eliminating round trips to the Cloud, as well as mask cloud computing outages. A key challenge for systems designed for edge computing is an efficient data management in the context of mobile devices and sensors/actuators which generally have stringent requirements on energy consumption as well as memory, CPU usages and network bandwidth.

Our prototype system, SuccinctEdge[1], has been designed for edge computing from the get go and adopts the Resource Description Framework (RDF). The adoption of this data model is motivated by the data integration and reasoning facilities it provides. Considering the former, the Linked Data principles[2] together with the large set of Knowledge Graphs (KGs) available via the Linked Open Data initiatives[3] ease the design of Internet of Things (IoT) applications. For instance, ontologies such as the Sensor, Observation, Sample, Actuator (SOSA [4]), Quantities, Units, Dimensions, and Types (QUDT) [5] or Smart Applicances Reference (SAREF)[6] considerably simplify the task of describing, manipulating and connecting sensors and actuators. These ontologies also serve smart measure management when reasoning services are introduced in SPARQL queries to infer implicit consequences from explicitly represented knowledge.

SuccinctEdge favors a compressed, single index storage approach to a solution based on multiple indexes that could potentially improve query execution but at the cost of a higher memory footprint. The applications we are targeting with SuccinctEdge are the processing of a flow of RDF graphs (sent from sensors or actuators) which are sharing a common topology. These graphs are continuously queried by a set of SPARQL queries. In a typical use case, these queries are searching for anomalies occurring over a network of sensors (see Section sec:example for a motivating example). As a result, these queries are executed once per graph instance.

Our system makes an intensive use of succinct data structures (SDS)[8], a family of data structures that adopts a compression rate close to theoretical optimum, but simultaneously allows efficient decompression-free query operations on the compressed data. Together with our single index approach, SDS guarantees a low memory footprint that fits with an in-memory storage approach. The decompression-free aspects also tends to reduce the number of CPU cycles on standard queries and inferences.

SuccinctEdge's reasoning services are based on the LiteMat encoding solution[4]. This approach prevents inference materialization and reduces the cost of the SPARQL query rewriting task, the two most frequent reasoning solutions in RDF stores. As a result of encoding most triple entries with integer values, this approach improves the efficiency of graph pattern matching and compresses RDF data sets, thus limiting the memory footprint of a given graph.

SuccinctEdge is addressing the compact storage and efficient querying of RDF data via SPARQL queries in the presence of RDFS reasoning in an edge computing environment. The main contributions of this paper are to (i) present a self-index, compact, in-memory storage layout based on the bitmap and wavelet tree SDSs, (ii) propose a decompression-free (*i.e.*, the SDS compressed graph does not need any decompression step to enable query execution), efficient query processing and optimization of SPARQL BGPs which are transformed into access, rank and select SDS operations, (iii) support reasoning during query processing using a smart encoding approach and (iv) propose a simple and automatic approach to express complex queries requiring inferences by preventing end-users to learn the details of used ontologies and ontology annotations used at each sensor.

We demonstrate the efficiency of our implementation on an evaluation conducted on real-world and synthetic data sets. This paper is organized as follows. In Section 2, we motivate our approach with a real-world example in an industrial setting. In Section 3, we provide some background knowledge. Section 4 presents the overall architecture of SuccinctEdge. The query optimizer and processor is presented in Section 5. Section 6 relates our research to existing work and Section 7 provides a detailed experimentation. We conclude the paper and present directions for future work in Section 8.

2 MOTIVATING EXAMPLE

In this paper, we consider an upcoming deployment of SuccinctEdge at some of ENGIE's buildings where an IoT network is

[1]https://github.com/xwq610728213/SuccinctEdge
[2]https://www.w3.org/wiki/LinkedData
[3]https://lod-cloud.net/
[4]http://www.w3.org/TR/ns/sosa
[5]http://qudt.org/schema/qudt
[6]https://ontology.tno.nl/saref.ttl

deployed. ENGIE is a multinational company operating in fields such as energy transition, generation and distribution.

Our running example focuses on data harvested from a building management system with a first focus on potable water distribution. Intuitively, a flow of measures are obtained from a network of sensors. A thorough analysis permits to detect anomalies such as leaks or other abnormal situations from, for instance, pressure and flow measurements. The measures are usually represented as text files (*e.g.*, CSV) but, thanks to some mapping assertions and dedicated digital services deployed through APIs, are transformed into a form of RDF graph (to be detailed later in this paper) and annotated with concepts and properties of a domain ontology.

Figure 1 presents an extract of such a graph which concerns pressure and chemistry measures related to the water distribution management. Given such graph instances, our SuccinctEdge system executes queries that can detect some patterns such as anomalies linked to the water management system, *e.g.*, incorrect chemistry properties, network leak, etc. In a non edge computing context, each measure would transit on a computing network to a more powerful machine that could process the anomaly detection. Such an approach as several drawbacks: (i) it makes an intensive use of the computing and communication network which can rapidly be overloaded, *e.g.*, devices on the edge of the network generally have low bandwidth, (ii) the high-end computing machine also risks to be overcharged and stressed from the amount of data received (potentially from hundreds to thousands of sensors) and (iii) sending these data packets over the network is not cost-free for these sensors, *e.g.*, in terms of energy consumption.

In a context where anomalies are the exception, it makes sense to detect anomalies as close as possible to the sensors since it would require to (i) send fewer data over the computing network as that would occur only in anomaly cases, (ii) reduce decision latency and (iii) keep the high-end computing machine unstressed.

In our experimentation at ENGIE, we are designing a query-based anomaly detection approach that does not require from the end-users a high level of expertise on the underlying domain ontologies and its reasoning services. Hence these users only express queries in relatively high concept terms and do not have to worry about the inferences which are handled automatically by the system. Expressing a query with abstract concepts, *i.e.*, high in the concept hierarchy, permits to write a single query that can tackle sensors performing similar measures but annotated with different concepts and possibly with different measure units. This is an important requisite for our use case where different sensor brands and types can coexist in a given network. The simplicity of this approach was expected from ENGIE for productivity reasons. In fact, it enables its sensor personnel to concentrate on their tasks and not on adapting a given query to the potentially large number of sensors in an industrial setting. For instance, in the following real-world example, 2 sensor platforms are measuring similar values, *e.g.*, pressure and chemistry-related, but each sensor annotates them with different concepts. Considering Station1 the pressure and chemistry are respectively annotated with $qudt : PressureOrStressUnit$ and $qudt : Chemistry$, while for Station2, it is resp. $qudt : Pressure$ and $qudt : AmountOfSubstanceUnit$. Moreover, the pressure value in Station1 is expressed in Bar while it is measured in HectoPascal in Station2.

Since, the QUDT ontology[7] states that:

$qudt : AmountOfSubstanceUnit \sqsubseteq qudt : Chemistry \sqsubseteq qudt : ScienceUnit$ and $qudt : PressureOrStressUnit \sqsubseteq qudt : PressureUnit \sqsubseteq qudt : MechanicsUnit$, a single SPARQL query can be written to address the peculiarities of each sensor at these 2 stations. The following query detects anomalies related to an incorrect pressure value (either expressed in Bar or HectoPascal) for sensors of stations 1 and 2:

```
SELECT ?x ?s ?ts ?v1 WHERE {
?x a sosa:Platform; sosa:hosts ?s.
?s sosa:observes ?o; a sosa:Sensor.
?o sosa:hasResult ?y; a sosa:Observation;
sosa:resultTime ?ts. ?y a sosa:Result;
qudt:numericValue ?v1; qudt:unit ?u1.
?u1 a qudt:PressureUnit. FILTER (?newV<3.00 || ?newV>4.50)
BIND(if(regex(str(?u1),"http://qudt.org/vocab/unit/BAR"),?v1,
if(regex(str(?u1),"http://qudt.org/vocab/unit/HectoPA")
,?v1/1000,0)) as ?newV) }
```

3 BACKGROUND KNOWLEDGE

3.1 Semantic Web standards

RDF is the W3C recommendation schema-free data model that supports the description of data on the Web. It takes the form of a graph consisting of a set of triples. Each triple is composed of a subject, a predicate and an object. Properties can be qualified as object or datatype. They both related a URI (or blank node) to respectively a URI (or blank node) and a literal. SPARQL, another W3C recommendation, enables to express queries over RDF data. The syntax is inspired by SQL's SELECT-FROM-WHERE but it uses an approach based on matching a BGP, *i.e.*, a set of triple patterns (TP), on an RDF graph to retrieve query answer sets. Finally, RDF Schema (RDFS) and Web Ontology Languages (OWL) enable the description of vocabulary semantics used in RDF data sets. They support inference services based on their respective expressiveness.

3.2 LiteMat

LiteMat is a semantic-aware encoding scheme that compresses RDF data sets and supports reasoning services associated to the RDFS ontology language. In this work, we focus on the ρdf[7] subset of RDFS, *i.e.*, inferences associated to the `rdfs:range`, `rdfs:domain`, `rdfs:subClassOf` and `rdfs:subPropertyOf` properties. To address inferences drawn from these last two RDFS predicates, we attribute numerical identifiers to ontology terms, *i.e.*, concepts and predicates, that are supporting the semantics. This is performed by prefixing the encoding of a term with the encoding of its direct parent. This encoding is computed using a binary representation and all binary encoding entries are all of the same length. The encoding is performed using a top-down approach, *e.g.*, starting from the most specific concept of the hierarchy (typically `owl:Thing`, `owl:topObjectProperty` and `owl:topDataProperty` for respectively the concept, object property and datatype property hierarchies), until all leaves are processed. Then a normalization is performed to guarantee that all encoding entries have the same length, *i.e.*, by setting right-most bits to 0.

We now provide an example on a concept hierarchy (a similar approach is used for property hierarchies). In Figure 2, we consider a small ontology extract containing the following axioms: $A \sqsubseteq Thing$, $B \sqsubseteq Thing$, $C \sqsubseteq B$ and $D \sqsubseteq B$. Figure 2a

[7] https://qudt.org/

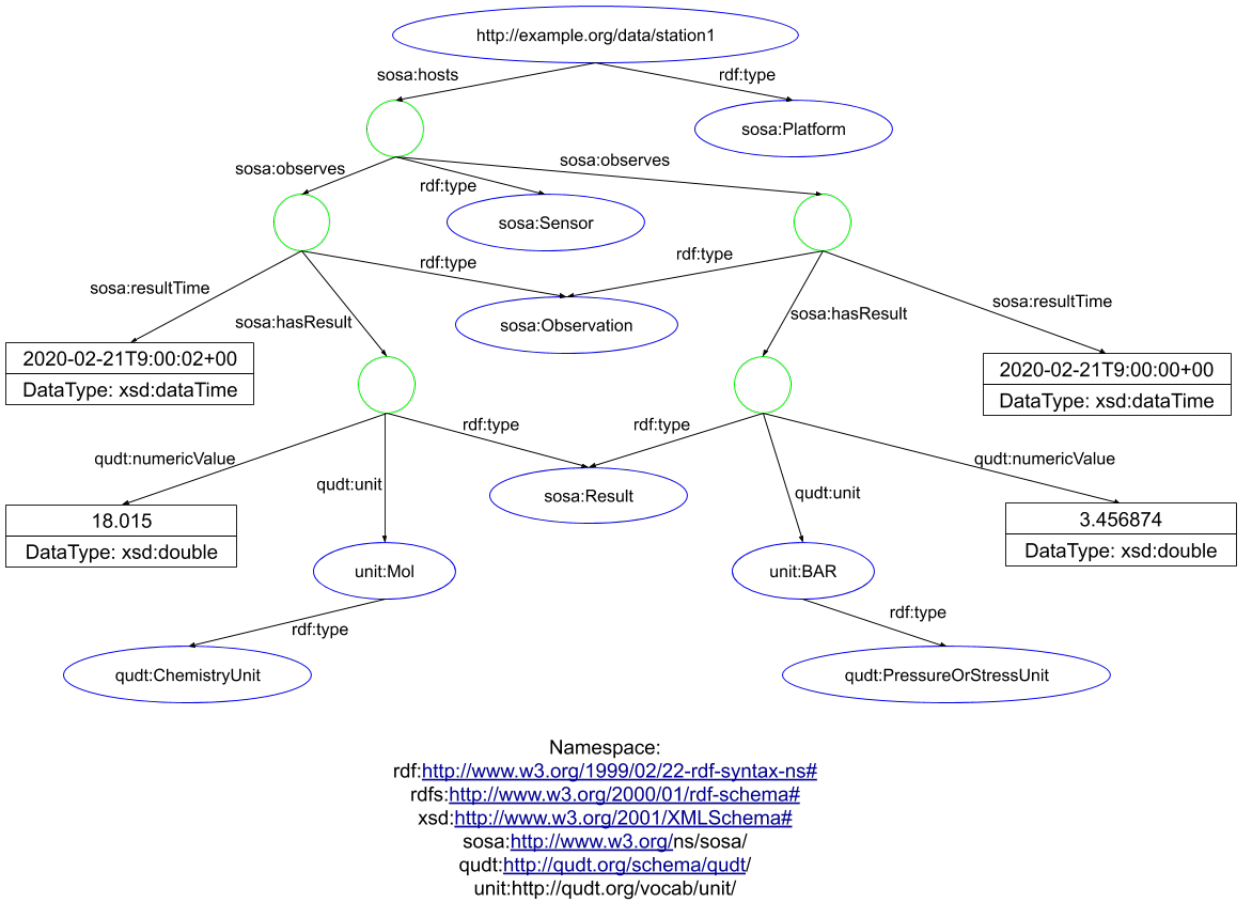


Figure 1: Graph extract of our use-case (green nodes are blank nodes)

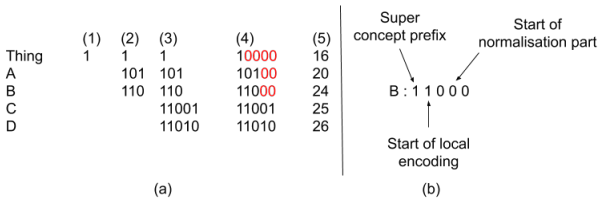


Figure 2: LiteMat encoding example

highlights the top-down encoding approach with (1) setting the local identifier of *Thing*, (2) its direct sub-concepts (*A* and *B*) and *B*'s sub-concepts in (3). Then, in (4) the normalization step is performed, *i.e.*, added right-most bits are written in red. Column (5) provides the integer values attributed to each concept.

The mapping between URIs and their identifiers are stored in dictionaries, two for the concepts and two for the properties to support a bidirectional retrieval, *i.e.*, from a URI to its identifier and from an identifier to its URI. Moreover, in the former dictionaries, additional identifier metadata are stored. For instance, the local length (binary length before the normalization phase) of each dictionary entry is stored along the final identifier entry.

Figure 2(b) emphasizes the different metadata of the LiteMat encoding for the *B* concept: super concept identifier part, start of local encoding and start of the normalization part.

The semantic encoding of concepts and predicates supports reasoning services usually required at query processing time. For instance, consider a query asking for the pressure value of sensors of type `S1`. This would be expressed as the two following TPs: `?x pressureValue ?v. ?x type S1`. In the case sensor concept `S1` has n sub-concepts, then a naive query reformulation requires to run the union of n+1 queries. With LiteMat's semantic-aware encoding, we are able, using two bit-shift operations and an addition, to compute the identifier interval, *i.e.*, [lowerBound, upperBound), of all direct and indirect sub-concepts of `S1`. And thus we can compute this query with a simple reformulation: (i) replacing the concept `S1` with a new variable : `?x type ?newVar` and (ii) introducing a filter clause constraining values of this variable: `FILTER (?newVar>=lowerBound && ?newVar<upperBound)`.

Considering the instance dictionary, each distinct entry is assigned an arbitrary unique integer value.

3.3 Succinct Data structures

SDS represents a family of data structures that stores data in a compact way, but still allows some efficient data access operations without decompression. There are different types of SDS, among which we consider Wavelet Tree (WT) and BitMap (BM).

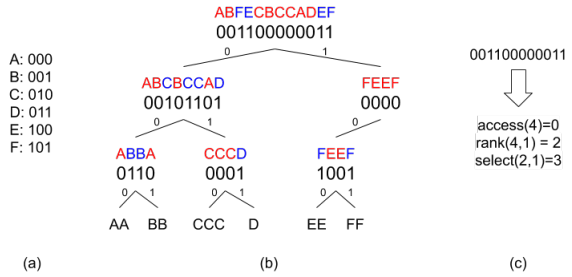


Figure 3: Wavelet Tree example with its dictionary

SuccinctEdge represents an RDF graph into a combination of these two structures to reach a very compact size without loss of query efficiency.

BM is the most basic SDS we are using in SuccinctEdge. It is a sequence of bits with some extra information to support the efficient execution of SDS operations. BM is the basic building block of WT's nodes (as each node in the tree is a BM), but it also relates different WTs in SuccinctEdge's triple representation (further details in Section 4).

WT [8], whose name reveals some affinity with the idea of the wavelet packet decomposition in signal processing, refers to a data structure which decomposes a data sequence into a set of nodes of a balanced binary tree. An example of a WT is given in Figure 3b. Suppose that we have a sequence *ABFECBCCADEF*, where each letter is mapped with an identifier in an incremental order, *e.g.*, *A* is denoted with 0, *B* is denoted with 1 (see dictionary in Figure 3a. A tree structure is constructed from this sequence as follows: each level of this tree divides the sequence of previous nodes into two sub-sequences by the corresponding bit. For example, from root to the first level, *ABFECBCCADEF* is divided into *ABCBCCAD* and *FEEF* by the first bit of each identifier entry. This strategy is applied recursively until each leaf is computed.

SDS support three operations to access data: *Rank*, *Select* and *Access*. Given a sequence *S*, the operation $S.Access(i)$ (also denoted as $S[i]$) refers to the $(i+1)^{th}$ element in *S*. $S.Rank(i,c)$ returns the number of occurrences of *c* from *S*'s beginning to index *i*. Finally, $S.Select(i,c)$ returns the index of i^{th} occurrence of element *c* in *S*. These operations can be computed in O(1) for BM and O(log n) for WT where n is the size of the vocabulary. Figure 3c provides an example over a simple BM. Dedicated algorithms permit to compute these 3 operations over WT.

4 ARCHITECTURE OVERVIEW

Before providing an overview of the SuccinctEdge RDF store, we describe a standard running setting at an ENGIE building. Typically, the person responsible for the building maintenance supervises a set of IoT devices from a SuccinctEdge server. From this central computer, the administrator is able to register new IoT devices installed in this set of buildings. Each IoT device typically runs a SuccinctEdge instance (client) which can execute many SPARQL queries. The administrator receives alerts from SuccinctEdge instances has abnormal sensor measures are occurring. Hence, each sensor modification (*e.g.*, a sensor is replaced due to a failure, a sensor data schema is modified) must go through an administration step which is performed on a central computer. Apart from such maintenance operations, this server also performs the pre-processing task consisting of encoding ontologies using the LiteMat scheme. In this context, and we consider in a large number of industrial settings, the ontologies are stable and thus rarely change. As explained previously, in SuccinctEdge, these ontologies take the form of a set of dictionaries (since their semantics are encoded via the use of LiteMat). These dictionaries are broadcasted to the different SuccinctEdge instances running at the edge.

An overview of SuccinctEdge's architecture is presented in Figure 4. Like most RDF stores, all triples are encoded according to some dictionaries. The underlying basic concept of a dictionary is to provide a bijective function mapping long terms (*e.g.*, URIs, blank nodes or literals) to short identifiers (*e.g.*, integers). More precisely, a dictionary should provide two basics operations : `string-to-id` and `id-to-string` (also referred in the literature as `locate` and `extract` operations). In a typical use of SuccinctEdge, the query engine will call the `locate` operation to rewrite the query into a list to match the data encoding, while the `extract` operation will be called to translate the result into the original format. In our case, we are using LiteMat (see Section 3.2) to generate the concept, property and individual dictionaries.

The Triple store component adopts a single index based on the predicate, subject, object (PSO) triple permutation. That is, the triples of the graph are sorted in ascending order over the P, S and O values of our dictionaries. The PSO order is motivated by the fact that the basic graph pattern of queries submitted to SuccinctEdge have predicates filled in with URIs (as opposed to variables). This corresponds to typical IoT use cases where queries are retrieving information from measures rather than serving to discover patterns in the graphs. In fact, there is no need for discovery since the graph patterns are well known in advance and are very rarely modified (*i.e.*, mostly due to sensor failure in industrial use-cases).

The Triple store component also highlights that we make a distinction between object (expect `rdf:type`) and datatype properties. In the former, objects are individuals and thus encoded with the respective instance dictionary while in the latter, objects are literals and stored using a flat data structure to store literals. This last data structure is motivated by the fact that it is not reasonable to create an entry in the instance dictionary for each new literal value. Intuitively, a sensor generally sends numerical values corresponding to physical measurement at a given time. Depending on the precision of these measures, the amount of different values to store in the instance dictionary is potentially infinite. So, we prefer to store the values as they have been sent by sensors, possibly with some redundancy, in order to prevent a complex and costly individual dictionary management.

In terms of data structures, WTs are used for the property and subject layers as well as the object layer for object properties. In order to relate a WT of one layer to another, we are using a BM. Figure 5b represents the triple set of Figure 5a where a WT corresponds to balanced tree of BMs. Intuitively the PS (respectively SO) bitmap permits to link a given P (resp. S) to several S (resp. O) values. In Figure 5b, p1 is connected to s1, s2 and s4 because the PS bitmap starts with a 100 sequence: '1' states that the sequence of p1 starts with a given subject (s1) and the '00' states that 2 other subjects are linked to p1. Moreover, the 4th bit in the PS BM (*i.e.*, set to '1') starts the sequence of the second property entry in the P WT (*i.e.*, p2).

Finally, triples containing a `rdf:type` property are stored in the RDFType store layout. These triples generally represent an important proportion of the triple set in real-world RDF data sets. We simply store them in a red-black tree in order to maintain the

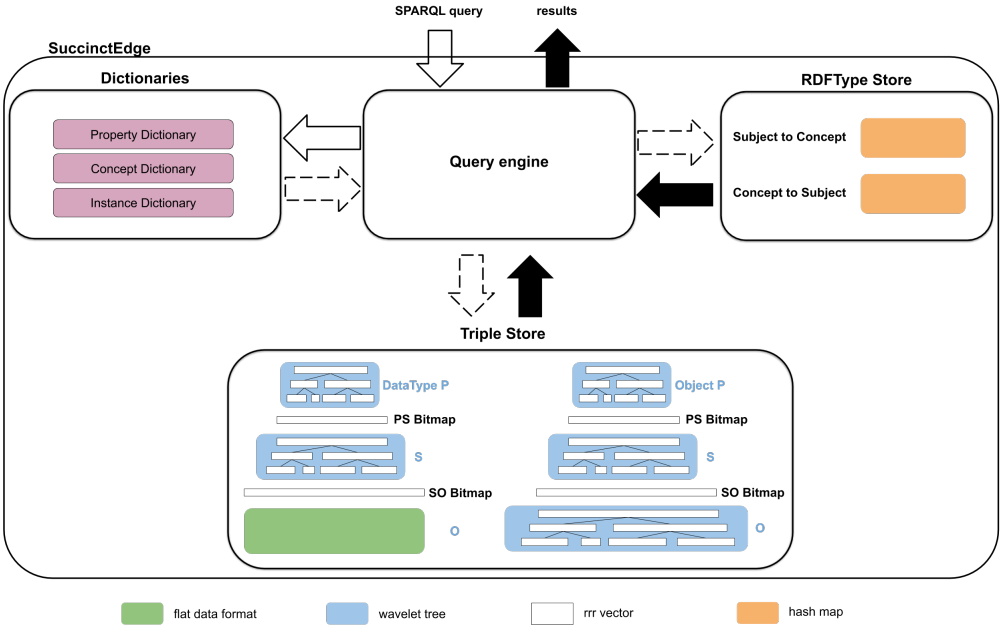


Figure 4: Architecture overview of SuccinctEdge

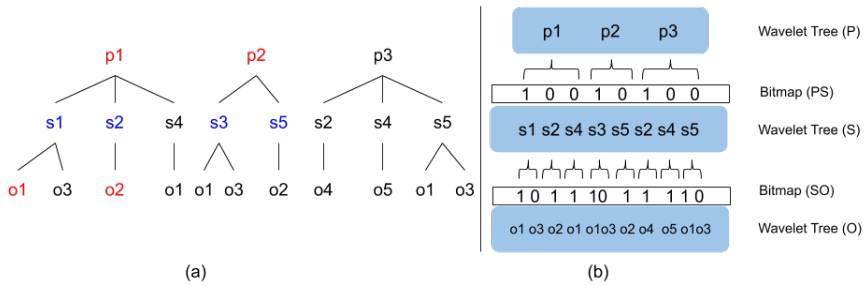


Figure 5: RDF graph representation: (a) as a PSO-based forest and (b) in SuccinctEdge as a combination of wavelet trees and bitmaps (only considering object properties)

search complexity to O(log(n)) while being fast when we insert `rdf:type` triples during database construction.

5 QUERY PROCESSING AND OPTIMIZATION

In this section, we present the query optimization and processing solutions developed for SuccinctEdge. Their main goals are respectively to define an efficient TP join ordering, by combining heuristic and cost-based approaches, and to generate a physical plan composed of SDS operations (*i.e.*, access, rank and select).

5.1 Query Optimization

The design of our query optimizer considers the limitations of the devices on which SuccinctEdge is running on, *i.e.*, limited memory space and computing power. Due to these constraints, our system only generates left-deep join trees which generally reduce the amount of memory used by the search process.

As stated in [10], join ordering is the most crucial issue in SPARQL query optimization. This is mainly due to the potentially high number of triple patterns and thus of join operations that one can find in BGPs. For instance, in our IoT building management experimentation, we have frequently encountered queries in the range of 10 joins.

In order to optimize a given SPARQL query, our query engine constructs a query graph where each TP of the SPARQL query corresponds to a node of the query graph. Each query graph node is also annotated to state whether its property is rdf:type or not. The nodes in this graph are connected if they share a common variable, hence forming a join. Moreover, the edges of this query graph are labeled with a join type, either SO or SS for respectively subject-object and subject-subject joins.

Example 5.1. Figure 6b displays the query graph associated with the SPARQL query presented in Figure 6a. This query contains 7 TPs, denoted tp1 .. tp7. The dotted nodes in the query graph correspond to rdf:type TPs.

Given a query graph, our optimizer uses Algorithm 1 to produce a join order. Intuitively, starting from a given TP, it invokes an overloaded *getMostSelective* method to search for the next TP to join with. This method uses a set of static rules together with some data statistics. In terms of the former, we have been influenced by Heuristic 1 of [12] which defines an execution

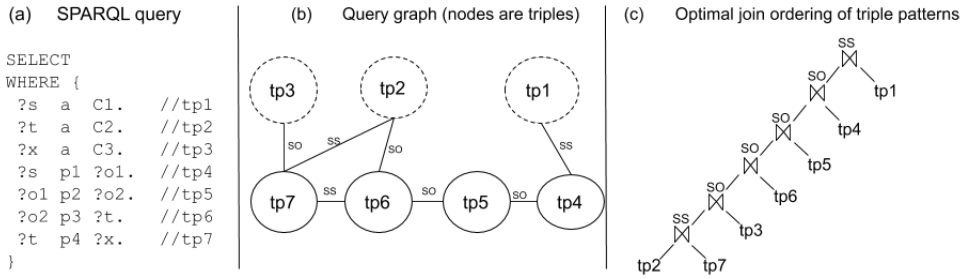


Figure 6: Query, query graph and join ordering

order for the 8 possible TP combinations. In the context of SuccinctEdge, we do not need to consider all combinations since TPs with either zero or three variables, *i.e.*, (s, p, o) and $(?s, ?p, ?o)$, are highly unlikely to occur in a real-world IoT SPARQL query. Intuitively, this heuristic states that TPs with the fewest variables should be executed first. Our adaptation re-orders the original proposition by taking into account the fact that our access paths are limited to PSO for non `rdf:type` properties and to SO/OS paths for `rdf:type` triples. As presented in Section 4, the latter access path (SO/OS on `rdf:type`) is more efficient than the one based on the SDS structures. Our TP order is thus:

(s,rdf:type,?o)>(?s,rdf:type,o)>(s,p,?o)>(?s,p,o)>(?s,p,?o), where p denotes any property different from rdf:type and the relation tp1 > tp2 states that tp1 should be executed before tp2. The (s,p,?o)>(?s,p,o) order is due to the navigation mode in our multi-layer SDS triple representation which is PSO based, *i.e.*, it is more efficient to retrieve objects given a subject/property pair than to compute subjects from a property/object pair. The (?s rdf:type ?o) TP is not considered relevant in a practical IoT context.

This first heuristic is generally not sufficient to decide which TP to execute first among a set of other TPs. Hence, we are considering a second heuristic that takes into consideration the linearity required by a left-deep join tree and examines the types of join possible between TPs. Due to the PSO self-index SDS structure used for non-rdf:type triples, SS joins are preferred over SO joins, *i.e.*, $S \bowtie S > S \bowtie O$. Other forms of joins, *i.e.*, SP, OP, PP have a lower priority since they are rarely encountered in the setting where SuccinctEdge is relevant.

In order to minimize intermediate results, the optimizer also relies on a set of statistics computed at dictionary creation-time. Intuitively, each dictionary persists the number of occurrences of each of its entries, *i.e.*, concept, property and non-literal individuals. Our statistic approach considers the hierarchy position of a given concept or property when computing the total number of triples it is involved in. For example, with the following concept hierarchy $C_2 \sqsubseteq C_1 \sqsubseteq C_0$ and $C_3 \sqsubseteq C_0$, the set of triples involving instances of concept C_0 will be the set of instances of type C_i with i $\in (0, 1, 2, 3)$. A similar process is applied to get the correct statistics for properties involved in a property hierarchy. Finally, some statistics are also computed at run-time, *e.g.*, the BM and WT data structures facilitate the computation of certain statistics. For instance, Algorithm 2 computes the number of triples containing a certain property.

Algorithm 1 first starts with the identification of the most selective rdf:type TP with an SS join. In the case it does not find an rdf:type TP or finds only rdf:type TP connected with SO joins, it then selects a non-rdf:type TP to start with. In the case several TPs satisfy our constraint, the statistics permit to take a decision. That first TP is appended to our *tpOrder* sequence. We then loop over the remaining nodes of the query graph until all TPs have been added to the sequence. At each iteration of the loop, the *getMostSelective* method considers TPs in the *tpOrder* sequence and searches for the next TP to append to this sequence. This search is again based on our two heuristics and the usage of statistics.

Example 5.2. The left-deep join tree displayed in Figure 6c has been defined using Algorithm 1 considering that tp2 is more selective than tp1, *i.e.*, the number of occurrences of C2 is lower than the one of C1. Once tp2 has been selected, the optimizer has the choice to join it with tp6 or tp7. tp7 is chosen since a SS join is preferred to a SO join. At this stage, the number of occurrences of concept C3, *i.e.*, tp3, can be lower than the number of already computed binding for ?x, and thus tp3 is selected. Given that tp2, tp7 and tp3 have already been considered, tp6 is the only alternative that can be considered and similarly for the remaining TPs, *i.e.*, tp5, tp4 and tp1.

Algorithm 1: Computation of a TP order

Input: query graph G
Output: ordered sequence of TPs

```
tpOrder ← ∅;
n ← getMostSelective(rdf : type);
tpOrder ← tpOrder + n;
while not all G nodes are in tpOrder do
    n ← getMostSelective(tpOrder);
    tpOrder ← tpOrder + n;
end
return tpOrder;
```

5.2 Query processing

Once an order is defined by SuccinctEdge's query optimizer, our system translates TPs into SDS's standard operations: access, rank and select. We are using an additional function, namely $rangeSearch(a, b, c)$, which finds all the occurrences of value c in the interval (a, b). It uses a binary search, *i.e.*, due to the ordering imposed on subjects for a given property, and returns the indexes of matching values. The use of this function speeds up query execution since it efficiently prunes searches by just computing the boundaries of the Subject WT, *i.e.*, first and last subject values of a given property, instead of scanning all values of that interval. A similar optimization is used when searching objects of given property/subject pair, *i.e.*, using the boundary of Object WT.

Algorithm 2: Compute the number of triples corresponding to a certain predicate.

Input: Predicate p
Output: Number n

$id_p \leftarrow FindIdFromDictionary(p)$;
$index_p \leftarrow wt_p.select(1, id_p)$;
$index_{sBegin} \leftarrow bitmap_{ps}.select(index_p + 1, 1)$;
$index_{sEnd} \leftarrow bitmap_{ps}.select(index_p + 2, 1)$;
$index_{oBegin} \leftarrow bitmap_{so}.select(index_{sBegin} + 1, 1)$;
$index_{oEnd} \leftarrow bitmap_{so}.select(index_{sEnd} + 2, 1)$;
$n \leftarrow index_{oEnd} - index_{oBegin}$;
return n;

We now present two translation examples in Algorithm 3 and 4 for resp. the $(s, p, ?o)$ and $(?s, p, o)$ TPs. Algorithm 3 shows how to retrieve an answer set with a $(s, p, ?o)$ TP. The idea is to first compute an interval of object values related to a given predicate and subject pair. This is performed by navigating through our BM and WT structures. All the objects in this interval are the results of this TP. Algorithm 4 retrieves all the subjects of a $(?s, p, o)$ TP. Unlike Algorithm 3, we can not locate all the subjects directly. So our strategy is to get the interval of all the objects corresponding to the known predicate top-down, after which we locate the object in this interval (there may be multiple appearances) and get the corresponding subjects.

Algorithm 3: Search the triple pattern $(s, p, ?o)$

Input: Predicate s,p
Output: Results res

$id_p \leftarrow FindIdFromDictionary(p)$;
$id_s \leftarrow FindIdFromDictionary(s)$;
$index_p \leftarrow wt_p.select(1, id_p)$;
$index_{sBegin} \leftarrow bitmap_{ps}.select(index_p + 1, 1)$;
$index_{sEnd} \leftarrow bitmap_{ps}.select(index_p + 2, 1)$;
for $index_s$ *in* $wt_s.rangeSearch(index_{sBegin}, index_{sEnd}, id_s)$ **do**
  $index_{oBegin} \leftarrow bitmap_{so}.select(index_{sBegin} + 1, 1)$;
  $index_{oEnd} \leftarrow bitmap_{so}.select(index_{sEnd} + 2, 1)$;
  for $index_o \leftarrow index_{oBegin}$ **to** $index_{oEnd}$ **do**
    $id_o \leftarrow wt_o[index_o]$;
    add (id_s, id_p, id_o) into res;
  end
end
return res;

In cases where reasoning services are necessary to provide an exhaustive answer set, we can replace $index_p$ with a continuous interval corresponding to a LiteMat interval. This interval is efficiently computed given the order imposed on leaves of a certain WT, *e.g.*, Property WT for the property hierarchy. The larger and deeper a property hierarchy, the more efficient this optimization approach since it prevents from navigating in the complete tree of a given WT.

TPs containing `rdf:type` are processed differently using the RDFType store component, where some simple structure look-ups permit to efficiently retrieve to subjects of a given concept or the concepts of a given subject.

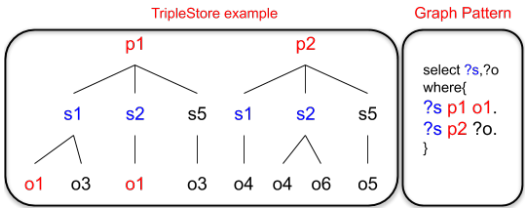


Figure 7: Merge join example

The next step corresponds to joining the results obtained from the execution of TPs. This occurs when different TPs share a common variable. One of our joining approach amounts to propagate variable assignments from one TP to another. Consider the triple set of Figure 5a and TPs $(?s, p1, o1)$ and $(?s, p2, ?o)$. The first TP gets the following assignments: $?s : \{s1, s2\}$ which will serve to dynamically generate $(s1, p2, ?o)$ and $(s2, p2, ?o)$ for the second triple.

During the join operation, we can benefit from a merge join (due to the original PSO value order) in certain cases when the values assigned to a joining variable to the TP are kept in order. For instance, in the case of a star-shaped BGP, *e.g.*, $(?s, p1, o1)$ and $(?s, p2, ?o)$, thanks to the facts that all the subjects connected to a certain predicate are ordered and that all the objects connected to one certain subject are also ordered, we can perform a merge join on the subject variable. Figure 7 provides a graph pattern (on the right side) and an RDF Graph (left side). From the first TP, we can retrieve $\{(p1, s1, o1), (p1, s2, o1)\}$ as the answer set. Clearly, since the subjects are ordered for a given predicate, the system can easily use a merge join with the 2nd TP of the query. In cases where the order is not guaranteed, we use nested loop joins.

Algorithm 4: Search the triple pattern $(?s, p, o)$

Input: Predicate p
Output: Results res

$id_p \leftarrow FindIdFromDictionary(p)$;
$index_p \leftarrow wt_p.select(1, id_p)$;
$index_{sBegin} \leftarrow bitmap_{ps}.select(index_p + 1, 1)$;
$index_{sEnd} \leftarrow bitmap_{ps}.select(index_p + 2, 1)$;
$index_{oBegin} \leftarrow bitmap_{so}.select(index_{sBegin} + 1, 1)$;
$index_{oEnd} \leftarrow bitmap_{so}.select(index_{sEnd} + 2, 1)$;
for $index_o$ *in* $wt_o.rangeSearch(index_{oBegin}, index_{oEnd}, id_o)$ **do**
  $index_s \leftarrow bitmap_{so}.rank(index_o + 1, 1) - 1$;
  $id_s \leftarrow wt_s[index_s]$;
  add (id_s, id_p, id_o) into res;
end
return res;

Previous executions steps are repeated until all the TPs have been processed. Then the answer set of the query is translated using our dictionaries and presented to the end-user or application.

6 RELATED WORK

Header Dictionary Triples (HDT)[6] is a compact data structure and binary serialization for RDF data. The Triples component of HDT requires that triples are sorted in a specific order, *e.g.*, SPO.

The triples are stored in so-called Bitmap Triples which represents a forest of RDF trees, *e.g.*, each tree is rooted with a given subject value. The remaining tree layers, *e.g.*, for P and O, each correspond to a sequence of identifiers and a bit sequence which connects layers like our BMs. Like HDT, SuccinctEdge represents RDF triples as trees but it makes an extensive use of WTs and depends on three different storage approaches, namely Object-triple-store, Datatype-triple-store and RDFType-store. Moreover, SuccinctEdge is equipped with a full-fledged query processing component and supports RDFS reasoning within SPARQL queries.

In [11], a so-called Semantic Index is proposed for Ontology-Based Data Access (OBDA) systems. In this approach, each entity (concept or property) in the corresponding hierarchy is assigned a numerical value according to a breadth-first search traversing of the hierarchy. Provided with this assignment, one is ensured that any sub-hierarchy is associated to a consecutive set of numerical values (*i.e.*, an interval). Intuitively, each entity is associated to an interval covering the indices of all its sub-entities. This approach is related to the LiteMat encoding scheme but the integration in SuccinctEdge permits a high compression rate with a decompression-free approach at query execution time. Moreover, Semantic Index is just an encoding scheme for a knowledge base and is not a complete query processor.

WaterFowl[3] was designed as a first attempt to use SDS for RDF storage and query processing. Although its compactness can be used in an edge computing setting, it lacks the different object storage implementation and query processing (including optimization) features of SuccinctEdge.

RDF4Led is an RDF store designed for edge computing. Compared to our system, RDF4led is disk-based, *i.e.*, it stores data on a SD card, and depends on multiple indexes which imply a high memory footprint. Moreover, it doesn't support reasoning services nor SPARQL's UNION clause which prevents to apply a query rewriting in order to support reasoning services.

ZipG[5] is a distributed graph store designed for the property graph data model. Hence it does not provide support for SPARQL (or any declarative query language) or reasoning services. Its storage layout is based on the Succinct[1] system and is mainly composed of flat binary unstructured files. ZipG is not compatible with devices located at the edge of a network. In fact, it thrives in a cloud computing setting.

TerminusDB[8] is an open-source general-purpose graph database. It aims to store very large graphs in main memory by scaling vertically. Such an approach is not compatible with the edge computing ecosystem that we are targeting. Moreover, TerminusDB does not support Semantic Web standards and hence can not benefit from the existence of a large set of available ontologies to support data integration or support reasoning services associated to RDFS or OWL ontology languages.

7 EVALUATION

7.1 Experimental setting

Our experimentation is conducted on a Raspberry Pi 3B+ which can be considered as a typical edge computing device on which we can run some sophisticated programs. This small computer is equipped with a Cortex-A53 (ARMv7l) 32-bit SoC 1.4GHz CPU and 1GB LPDDR2 SDRAM. A SD-card, a widely used memory solution on such devices, of 8GB is used as persistent storage.

Considering the evolution of technology, it is widely accepted that edge computing devices will be more and more powerful in the near future. Hence, it is quite obvious that devices with sufficient calculation power and memory, *e.g.*, Raspberry Pis, Odroids, etc. , will be deployed at the edge of networks.

SuccinctEdge is implemented in C++14 and the SDS-lite C++ library[9] is required during compilation. More details can be found on github[10]. We are comparing SuccinctEdge against RDF4Led[13], two Apache Jena[11] (version 3.15) database implementations and RDF4J's Memory Store[12] (version 3.4.0) . RDF4Led is to the best of our knowledge the only RDF store specifically designed for edge computing. It is characterized by a small memory footprint, although the database system does not reside in the main-memory. The two Apache Jena stores are TDB2 and the in-memory store. They are both open-source relatively lightweight and robust RDF stores. RDF4J (originally Sesame) is an open-source Java Framework for managing RDF data. The core RDF4J databases are mainly intended for small to medium-sized data sets and thus it makes sense to consider them for Edge computing. We also considered RDFox [9], a main-memory, centralized RDF store that is designed on a shared-memory architecture, but could not make it work on our raspberry Pi 3B+ since we only had access to a 64-bit pre-compiled version. Systems like Ontotext GraphDB[13], Stardog[14], MarkLogic[15], AllegroGraph[16], AWS Neptune[17] have not been considered since they have been designed for massive loads and scalability on high-end servers or Cloud computing.

7.2 Datasets and queries

The experimentation uses both synthetic and real-world data sets. This duality is motivated by the current lack of large graphs emitted from sensors at our industrial partner. In fact, our real-world data sets, which correspond to the water management distribution in ENGIE's building, consist of 250 and 500 triples. They are denoted with their number of triples in this experimentation.

Due to these size limitations, it is not possible to stress SuccinctEdge in terms of graph sizes. Hence, we are also experimenting with the synthetic Lehigh University Benchmark (LUBM)[18] which can be easily configured to produce large data sets. Starting from a LUBM with one university, *i.e.*, composed of over 103.000 triples (denoted 100K), we created several triple subsets of 1.000, 5.000, 10.000, 25.000 and 50.000 triples which are respectively denoted as 1K, 5K, 10K, 25K and 50K in the remaining of this section. They are used to evaluate the behaviors of the five evaluated data management systems. Note that some of these synthetic data sets have triple set size way beyond what most sensors are currently emitting in real-world industrial use-cases. All submitted queries are detailed in Section A and data sets are available on the system's Github page.

7.3 Experimentation results

In this section, we are aiming to compare the previously mentioned RDF stores (*i.e.*, Jena TDB, Jena in-memory, RDF4Led, RDF4J and SuccinctEdge) on the following dimensions: graph construction time, memory footprint (*i.e.*, the storage space taken

[8] https://terminusdb.com/

[9] https://github.com/simongog/sdsl-lite
[10] https://github.com/xwq610728213/SuccinctEdge
[11] https://jena.apache.org/
[12] https://rdf4j.org/
[13] https://www.ontotext.com/products/graphdb/
[14] https://www.stardog.com/
[15] https://www.marklogic.com/
[16] https://allegrograph.com/
[17] https://aws.amazon.com/fr/neptune/
[18] http://swat.cse.lehigh.edu/projects/lubm/

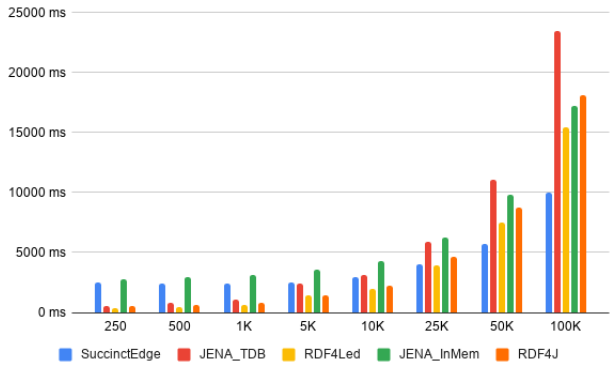


Figure 8: Construction time comparison

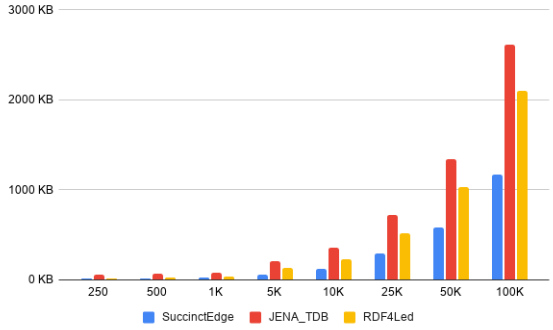


Figure 9: Dictionary size comparison

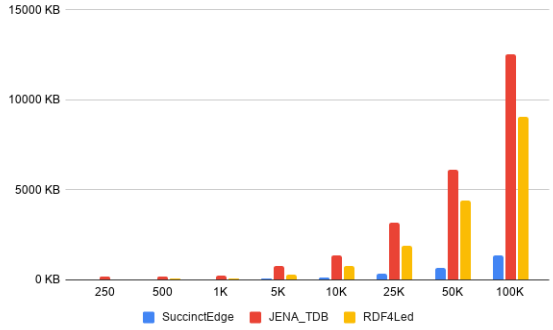


Figure 10: Storage size without dictionary comparison

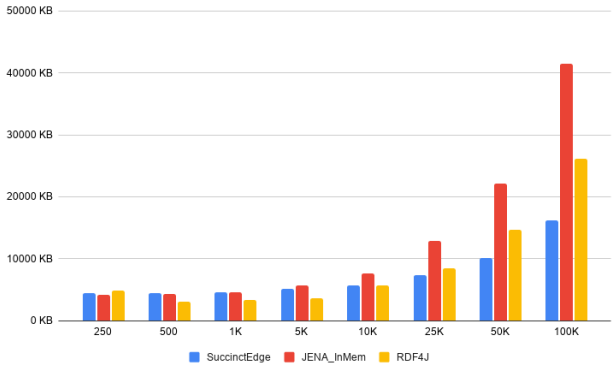


Figure 11: RAM footprint comparison

by different systems with the previous data sets), query execution performances on different triple patterns and basic graph patterns. Lastly, we evaluate the performance (duration time) of queries which necessitate reasoning services to produce an exhaustive answer set.

7.3.1 Back-end construction time. The back-end construction time corresponds to the time taken by each system to read the data set file and to construct its proper storage layout (including indexes in the case of all systems except SuccinctEdge which is self-index) on which queries can be asked.

In order to fully evaluate the performances of all the systems, we compare the back-end construction time of these systems with data sets ranging from 250 to over 100.000 triples. Figure 8 provides details on this experimentation. SuccinctEdge doesn't show much advantage when data set is rather small (up to 1.000 triples). We attribute this to the fact that the SDS-Lite library which is responsible for creating SuccinctEdge's BMs and WTs has an important start-up overhead that is relatively important compared to the effective duration of the structures. We consider that this may be optimized in future work, but it is out of the scope of this paper. However, as the data sets grow larger, our system outperforms all other systems.

7.3.2 Storage size. As SuccinctEdge is an in-memory RDF system, it is difficult to directly compare the memory occupation against Jena TDB and RDF4Led (which are both disk-based RDF stores). We persisted all the data structures existing in SuccinctEdge to disk in order to make a fair comparison.

We separately consider the dictionary and triple storage spaces. Figure 9 provides the three systems' dictionary sizes for all 8 data sets. In all cases, Jena TDB requires the largest memory footprint and SuccinctEdge takes about half of the size of RDF4Led.

Considering the triple storage space, displayed in Figure 10, SuccinctEdge consumes much smaller space thanks to its SDS-based storage implementation and self-index approach. This enables to reach one of our goal which is to store as much data as possible in a given amount of RAM.

We are also comparing the main-memory footprint of SuccinctEdge with the in-memory systems, *i.e.*, RDF4J and Jena_InMem. In this evaluation, it is not possible to distinguish between the space used for the dictionaries and the data sets. So we provide the total space amount. Figure 11 yields the experiment results. We can see that as the amount of data grows, SuccinctEdge gradually shows its strength in saving memory space. We mainly attribute this to the size of the indexes stored by both RDF4J and Jena_InMem.

7.3.3 Triple pattern query. Considering query processing, we start the evaluation with single triple patterns,*i.e.*, excluding the cost of join operations, in order to directly compare the performance of data retrieval in different systems.

We first consider the two interesting triple patterns containing a single variable in the context of SuccinctEdge: **S,P,?o** (queries S1 to S5) and **?s,P,O** (queries S6 to S10) . Moreover, we consider these two triple patterns with different selectivity, *i.e.*, result sets ranging from 4 to 521 tuples. Table 1 and 2 provide the results of this experimentation for the LUBM1 dataset (over 100.000 triples).

Table 1: Data retrieval for a single S,P,?o TP. The first row represents the number of triples in the answer set. All times in ms. Bold times are a column's most efficient.

	Query performance				
Query name	S6	S7	S8	S9	S10
Selectivity	4	66	129	257	513
SuccinctEdge	**0.3**	**3.5**	**6.2**	**10.9**	23.3
RDF4Led	12	28	33	47	84
Jena TDB	7	16	22	27	33
Jena_InMem	5	11	15	19	29
RDF4J	3	6	10	11.1	**13**

Table 2: Data retrieval for a single ?s,P,O TP. The first row represents the number of triples in the answer set. All times in ms. Bold times are a column's most efficient.

	Query performance				
Query name	S1	S2	S3	S4	S5
Selectivity	5	17	135	283	521
SuccinctEdge	**0.7**	**1.5**	**10.1**	20.7	32.0
RDF4Led	6	9	51	71	81
Jena TDB	7	8	30	32	41
Jena_InMem	7	8	15	21	27
RDF4J	3	3	11	**16**	**21**

As said previously, in an IoT setting, we are mainly interested in executing a query on the freshest data and such a query is generally execute only once per graph instance. Hence, we are only considering hot runs.

SuccinctEdge outperforms other systems on almost all query selectivity. It is only on relatively non-selective, at least considering an IoT context, that SuccinctEdge gets beaten by RDF4J (S4, S5 and S10). Considering our potable water distribution running example, the answer set of each query is clearly very selective. That is only a small set of tuples are retrieved from a specific query out of a given measure. We consider that this will be the case for many industrial situations. Thus, high selective queries is clearly the main playground for RDF stores running in Edge computing. In the case of selective queries, SuccinctEdge can be up to one order of magnitude faster than its RDF4J most direct competitor, *e.g.*, Table 1 S6 with a result set of size 4.

Figure 12 shows the results of several randomly picked **?s,P,?o** queries (triple patterns with a constant predicate and variable subject and object, denoted S11 to S15). We can see from the results that SuccinctEdge outperforms the other systems. Clearly, the conclusion obtained on single triple patterns with a single variable that the more selective, the more efficient SuccinctEdge is compared to the other systems, is confirmed. We attribute this to SuccinctEdge's in-memory approach and structure which is **?s,P,?o**-friendly due to its PSO self-index approach. Moreover Jena TDB and RDF4Led also have PSO or POS indexes but are disk-based database, for whom, loading data from disk takes a non-negligible time. The numbers of triples in the answer sets of our single variable TP experimentation are much smaller than that of the **?s,P,?o**. This leads to greater differences between the different systems. This is again due to the fact that RDF4Led and Jena TDB are loading data from disk. Nevertheless, we can consider that retrieving over 500 tuples at a time from a single sensor is already quite unusual for an IoT use case. The comparison

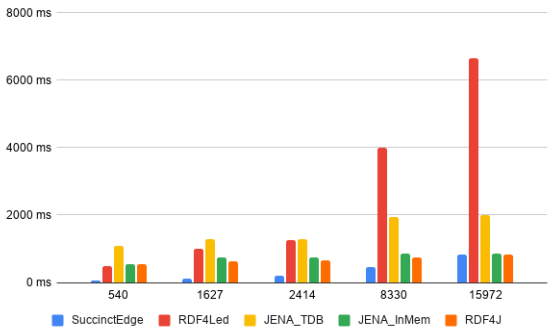


Figure 12: Data retrieval of queries with only one triple pattern of type ?s,P,?o, the x-axis represents the number of triples in the answer set.

with in-memory stores (RDF_InMem and RDJ4) highlights that SuccinctEdge is faster for answer sets lower than 10.000 tuples. At 16.000 result set tuples, The three systems behave similarly. Again, from the point of view of our experimentation partner, this is currently unusual for real-world industrial IoT use cases.

7.3.4 Graph pattern query. We now compare performances over queries containing multiple triple patterns, *i.e.*, requiring joins. Four queries with different selectivity values (answer sets ranging from 540 to close to 8.000 tuples) have been executed. They are denoted M1 to M5 and contain up to 10 TPs in the BGP. We can see in Figure 13 that RDF4Led and SuccinctEdge are always outperforming Jena TDB. SuccinctEdge is either more efficient than RDF4Led or slightly less efficient that RDF4Led. This showcases that in some cases RDF4Led finds a better TP query ordering strategy than SuccinctEdge and/or benefits from its large set of available indexes. Considering the latter, it is a price we are willing to pay for a lower memory footprint. Nevertheless, the former reason emphasizes that we can improve our query optimizer.

The comparison with the in-memory RDF stores emphasizes that the three systems behave similarly except for highly selective queries where SuccinctEdge is again more efficient. The differences between the query executions depend on the patterns used in the BGP of these five queries. Overall, we are satisfied that our system, with a single index, is at least at the same level than the two other systems.

7.3.5 Queries with RDFS reasoning. Our final experimentation concerns queries requiring some reasoning services. We have generated six queries (denoted R1 to R6) containing a mixture of RDFS:subClassOf and RDFS:subPropertyOf inferences. These queries present different selectivity characteristics, ranging from 15 to 8.345 tuples in the answer sets and contain up to 10 TPs in the BGP.

For SuccinctEdge, the reasoning service is automatically supported by LiteMat's encoding and is hence native in the system. This is not the case for the other systems for which we have rewritten each query as the union of all the possible sub-queries. Since RDF4Led doesn't support the SPARQL UNION clause no results are presented in Figure 14 for this system. Obviously, SuccinctEdge is much more efficient than Jena TDB. It is quite logical that the more entailments the query requires, the more efficient SuccinctEdge is compared to a system like Jena TDB.

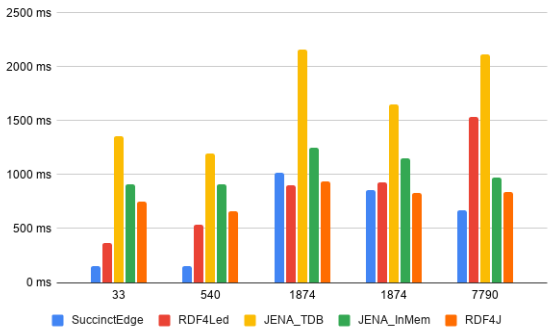


Figure 13: Queries with multiple triple pattern (x-axis corresponds to the number of tuples in the answer set)

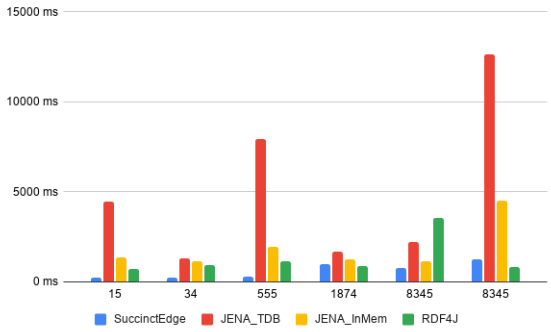


Figure 14: Queries with RDFS reasoning (x-axis corresponds to the number of tuples in the answer set)

As for Jena_InMem, it performs better than Jena TDB while still falling behind SuccinctEdge. When compared with RDF4J, SuccinctEdge performs better or similarly depending on the complexity of the reasoning services, *i.e.*, number of SPARQL UNION clauses. Note that we provide manual query rewriting to the Jena and RDF4J systems while these systems could implement the reasoning task with their APIs. In doing so, we provide a clear advantage to these systems since they do not have to load the ontology to perform the reasoning. Moreover, the extra cost of computing the UNION rewriting is not considered in the times of the Jena and RDF4J executions.

8 CONCLUSION

We have presented the first, to the best of our knowledge, KG inference-enabled data management system designed for Edge computing. Due to its unique index, compact, in-memory approach, we have demonstrated that SuccinctEdge outperforms its direct competitors on the following dimensions: query performance on different query patterns, efficiency of reasoning services, back-end size and creation time. The system is currently being deployed at some large building facilities at ENGIE and should help in detecting anomalies in water distribution and energy consumption. Due to its generic nature, SuccinctEdge is relevant for many IoT use cases such as anomaly and risk detection, supervising energy production and distribution. In the future, we are aiming to improve the query optimizer and support queries ranging several graphs. We are also considering to design a more efficient management of objects linked to datatype properties and to increase the expressiveness of supported ontology languages, *e.g.*, RDFS++ and OWL2RL. Moreover, we are considering the possibility of exchanging information with a larger graph portion that would reside in a cloud store.

REFERENCES

[1] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling Queries on Compressed Data. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*. 337–350. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/agarwal

[2] Yuan Ai, Mugen Peng, and Kecheng Zhang. 2018. Edge computing technologies for Internet of Things: a primer. *Digital Communications and Networks* 4, 2 (2018), 77 – 86. https://doi.org/10.1016/j.dcan.2017.07.001

[3] Olivier Curé, Guillaume Blin, Dominique Revuz, and David Célestin Faye. 2014. WaterFowl: A Compact, Self-indexed and Inference-Enabled Immutable RDF Store. In *The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings*. 302–316. https://doi.org/10.1007/978-3-319-07443-6_21

[4] Olivier Curé, Weiqin Xu, Hubert Naacke, and Philippe Calvez. 2019. LiteMat, an Encoding Scheme with RDFS++ and Multiple Inheritance Support. In *The Semantic Web: ESWC 2019 Satellite Events - Revised Selected Papers*. 269–284. https://doi.org/10.1007/978-3-030-32327-1_47

[5] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. 2017. ZipG: A Memory-efficient Graph Store for Interactive Queries. In *Proceedings of the 2017 ACM SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 1149–1164. https://doi.org/10.1145/3035918.3064012

[6] Miguel A. Martínez-Prieto, Mario Arias Gallego, and Javier D. Fernández. 2012. Exchange and Consumption of Huge RDF Data. In *The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings (Lecture Notes in Computer Science)*, Elena Simperl, Philipp Cimiano, Axel Polleres, Óscar Corcho, and Valentina Presutti (Eds.), Vol. 7295. Springer, 437–452. https://doi.org/10.1007/978-3-642-30284-8_36

[7] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. 2009. Simple and Efficient Minimal RDFS. *Web Semant.* 7, 3 (Sept. 2009), 220–234. https://doi.org/10.1016/j.websem.2009.07.003

[8] Gonzalo Navarro. 2014. Wavelet trees for all. *J. Discrete Algorithms* 25 (2014), 2–20. https://doi.org/10.1016/j.jda.2013.07.004

[9] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A Highly-Scalable RDF Store. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab (Eds.), Vol. 9367. Springer, 3–20. https://doi.org/10.1007/978-3-319-25010-6_1

[10] Thomas Neumann and Gerhard Weikum. 2009. Scalable join processing on very large RDF graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 627–640. https://doi.org/10.1145/1559845.1559911

[11] Mariano Rodríguez-Muro and Diego Calvanese. 2012. High Performance Query Answering over DL-Lite Ontologies. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*. AAAI Press, 308–318.

[12] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter A. Boncz. 2012. Heuristics-based query optimisation for SPARQL. In *15th International Conference on Extending Database Technology, EDBT , Proceedings*. 324–335. https://doi.org/10.1145/2247596.2247635

[13] Anh Lê Tuán, Conor Hayes, Marcin Wylot, and Danh Le Phuoc. 2018. RDF4Led: an RDF engine for lightweight edge devices. In *Proceedings of the 8th International Conference on the Internet of Things, IOT 2018*. 2:1–2:8. https://doi.org/10.1145/3277593.3277600

A QUERIES

A total of 26 queries have been evaluated over a LUBM data set consisting of over 100.000 triples. They can be dispatched into 2 groups: whether their contain a single triple pattern or multiple ones. In this section, we list only the most prominent queries and provide templates for the other ones. Moreover, we present their main characteristics. The interested reader can access all of them on the paper companion GitHub page[19]. The following prefixes apply to all queries: lubm <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>, rdf <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

[19] https://github.com/xwq610728213/SuccinctEdge

Table 3: Query summary with the following notations: 'SS' and 'OS' respectively correspond to subject, subject and object,subject joins; 'Co' for concept hierarchy inferences, 'Pr' for property hierarchy inferences

	Query performance													
Systems	S1-5	S6-10	S11-15	M1	M2	M3	M4	M5	R1	R2	R3	R4	R5	R6
TP number	1	1	1	2	3	5	3	11	5	5	3	6	3	11
TP type(s)	sp?	?po	?p?	?p?	?p? ?po	?p? ?po	?p? ?po sp?o	?p? ?po	?p? ?po	?p? ?spo	?p? ?po	?p? ?po	?p?	?p? ?spo sp?
Join type	-	-	-	SS	SS	SS,OS	OS	SS,OS OO	SS,OS	SS,OS	SS	SS,OS	OS	SS,OS OO
Join number	0	0	0	1	2	4	4	10	4	2	2	5	2	10
Path length	1	1	1	1	1	3	3	4	3	3	1	3	3	4
Selectivity	[4,513]	[5,521]	[540,15972]	540	1874	1874	7790	33	15	555	1874	1874	8345	34
Derived triples	0	0	0	0	0	0	0	0	15	540	1874	1874	555	1
Reasoning type	-	-	-	-	-	-	-	-	Co	Co Pr	Co Pr	Co Pr	Pr	Pr

A.1 Single triple pattern queries

This first set of queries contain a single triple pattern in the WHERE clause. We distinguish between queries with a single variable, either at the object (denoted sp?) or subject (denoted ?po) position, from queries with two variables (denoted ?p?). As explained in the paper, we do not consider that variables at the property position make sense in SuccinctEdge's use cases.

A.1.1 SP?o queries. The identification of these 5 queries range from S1 to S5. We used the following query template:

```
SELECT ?X WHERE {X1 P1 ?X}
```

For S1, P1 binds to the lubm:takesCourse property and X1 is an undergraduate student constant. For queries S2 to S5, P1 binds to lubm:publicationAuthor and the X1 bind to different publication instances. The selectivity of these queries are in Table 1.

A.1.2 ?sPO queries. These queries are identified from S6 to S10 and correspond to the following query template:

```
SELECT ?X WHERE { ?X P1 O1 }
```

P1 and O1 correspond to property and individual constants which for S6 to S10 respectively take the values (all properties are in the lubm namespace) : advisor/assistant professor constant, takesCourse/ course constant, worksFor/department constant, name/ publication constant, memberOf/ department constant.

A.1.3 ?sP?o queries.

```
S11: SELECT ?X ?Y ?Z WHERE { ?X lubm:worksFor ?Z }
S12: SELECT ?X ?Y ?Z WHERE { ?X lubm:teacherOf ?Y}
S13: SELECT ?X ?Y ?Z WHERE {
     ?X lubm:undergraduateDegreeFrom ?Y .}
S14: SELECT ?X ?Y ?Z WHERE { ?X lubm:emailAddress ?Y }
S15: SELECT ?X ?Y ?Z WHERE { ?X lubm:name ?Y }
```

A.2 Multiple triple patterns queries

In this set of queries, the BGP is composed of several triple patterns. The 11 queries in this category can be decomposed into those requiring or not some reasoning services (either based on concept or property hierarchies).

A.2.1 Non-inference queries. All prefixed with 'M'.

```
M1: SELECT ?X ?Y ?Z WHERE { ?X lubm:worksFor ?Z .
 ?X lubm:name ?Y .}
M2: SELECT ?X ?Y ?Z WHERE { ?X lubm:memberOf ?Z .
 ?X rdf:type lubm:GraduateStudent .
 ?X lubm:undergraduateDegreeFrom ?Y .}
M3:  SELECT ?X ?Y ?Z WHERE { ?X lubm:memberOf ?Z .
 ?X rdf:type lubm:GraduateStudent .
 ?Z rdf:type lubm:Department .
 ?Z lubm:subOrganizationOf ?Y .
 ?Y rdf:type lubm:University .}
M4: SELECT ?X ?Y ?Z WHERE { ?X lubm:memberOf ?Z .
 ?Z lubm:subOrganizationOf ?Y .
 ?Y rdf:type lubm:University }
M5: SELECT * WHERE {
 <http://www.Department0...Publication14>
 lubm:publicationAuthor ?p. ?st lubm:memberOf ?o2.
 ?p a lubm:AssociateProfessor. ?p lubm:worksFor ?o.
 ?o a lubm:department. ?o lubm:subOrganizationOf ?u.
 ?u a lubm:University. ?p lubm:teacherOf ?te.
 ?te a lubm:Course. ?st lubm:takesCourse ?te.
 ?st a lubm:UndergraduateStudent. }
```

A.2.2 Inference queries. The identifier of these queries is prefixed with an 'R' since they involve a form of reasoning.

```
R1: SELECT ?X ?Y ?Z WHERE { ?X rdf:type lubm:Person .
 ?Z rdf:type lubm:Department . ?X lubm:headOf ?Z .
 ?Z lubm:subOrganizationOf ?Y .
 ?Y rdf:type lubm:University .}
R2: SELECT ?X ?Y ?Z WHERE { ?X rdf:type lubm:Person .
 ?Z rdf:type lubm:Department .  ?X lubm:worksFor ?Z .
 ?Z lubm:subOrganizationOf ?Y .
 ?Y rdf:type lubm:University .}
R3: SELECT ?X ?Y ?Z WHERE { ?X lubm:memberOf ?Z .
 ?X rdf:type lubm:Student .
 ?X lubm:undergraduateDegreeFrom ?Y .}
R4: SELECT ?X ?Y ?Z ?N WHERE { ?X rdf:type lubm:Person .
 ?Z rdf:type lubm:Department . ?X lubm:memberOf ?Z .
 ?Z lubm:subOrganizationOf ?Y .  ?Y lubm:name ?N.
 ?Y rdf:type lubm:University . }
R5: identical to M4 but computes inferences  over the
memberOf property
R6: identical to M5 but computes inferences over the
memberOf and worksFor properties.
```

PolyFit: Polynomial-based Indexing Approach for Fast Approximate Range Aggregate Queries*

Zhe Li[1], Tsz Nam Chan[2], Man Lung Yiu[1], Christian S. Jensen[3]
Hong Kong Polytechnic University[1], Hong Kong Baptist University[2], Aalborg University[3]
richie.li@connect.polyu.hk,edisonchan@comp.hkbu.edu.hk,csmlyiu@comp.polyu.edu.hk,csj@cs.aau.dk

ABSTRACT

Range aggregate queries find frequent application in data analytics. In many use cases, approximate results are preferred over accurate results if they can be computed rapidly and satisfy approximation guarantees. Inspired by a recent indexing approach, we provide means of representing a discrete point dataset by continuous functions that can then serve as compact index structures. More specifically, we develop a polynomial-based indexing approach, called PolyFit, for processing approximate range aggregate queries. PolyFit is capable of supporting multiple types of range aggregate queries, including COUNT, SUM, MIN and MAX aggregates, with guaranteed absolute and relative error bounds. Experimental results show that PolyFit is faster and more accurate and compact than existing learned index structures.

1 INTRODUCTION

A *range aggregate query* [38] retrieves records in a dataset that belong to a given key range and then applies an aggregate function (e.g., SUM, COUNT, MIN, MAX) to an attribute of those records. Range aggregate queries are used in OLAP [38, 69] and data analytics applications, e.g., for outlier detection [72, 74], data visualization [24], and tweet analysis [54]. For example, network intrusion detection systems [74] utilize range COUNT queries to monitor a network for anomalous activities. Furthermore, applications with huge numbers of users are expected to receive queries frequently. For instance, Foursquare, with more than 50 million monthly active users [4], helps users find the number of specific POIs (e.g., restaurants) within given regions [3]. In many application scenarios, users accept approximate results provided that (i) they can be computed quickly and (ii) they are sufficiently accurate (e.g., within 5% error). We target such applications and focus on error-bounded evaluation of range aggregate queries.

A recent indexing approach represents the values of attributes in a dataset by continuous functions, which then serve to enable compact index structures [28, 44]. When compared to traditional index structures, this approach is able to yield a smaller index size and faster response time. The existing studies [28, 44] focus on computing exact results for point and range queries on 1-dimensional data. In contrast, we conduct a **comprehensive study of approximate range aggregate queries, supporting many aggregate functions and multi-dimensional data**.

The idea that underlies our proposal for using functions to answer approximate range aggregate queries may be explained as follows. Consider a stock market index (e.g., the Hong Kong Hang Seng Index) at different times as a dataset $\mathcal{D}$ consisting of records of the form (index value, timestamp), where the former is our measure and the latter is our key that is used for specifying query ranges—see Figure 1(a). A user can find the average stock market index value in a specified time range $[l_q, u_q]$ by issuing a range SUM query (and divide by $u_q - l_q + 1$). We propose to construct the cumulative function of $\mathcal{D}$ as shown in Figure 1(b). If we can approximate this function well by a polynomial function $\mathbb{P}(x)$ then the range SUM query can be approximated as $\mathbb{P}(u_q) - \mathbb{P}(l_q)$, which takes $O(1)$ time. As another example, the user may wish to find the maximum stock market index in a specified time range. The timestamped index values in $\mathcal{D}$ can be modeled by the continuous function shown in Figure 1(c). Again, if we can approximate this function well using a polynomial function $\mathbb{P}(x)$ then the range MAX query can be answered quickly using mathematical tools, e.g., by applying differentiation to identify maxima in $\mathbb{P}(x)$.

Regarding the two-dimensional case, consider the dataset of tweets' locations as shown in Figure 9(a) in Section 6, where each data point has a longitude (as key 1) and a latitude (as key 2). Suppose that the user wishes to count the number of tweets in a geographical region. Our idea is to derive the cumulative count function shown in Figure 9(b), and then approximate this function with a polynomial function $\mathbb{P}(x_1, x_2)$ (of two variables). This enables us to answer a two-dimensional range COUNT query in $O(1)$ time.

Another difference between our work and existing studies [28, 44] is the types of functions used for approximation. Our proposal uses piecewise polynomial functions, rather than piecewise linear functions [28, 44]. As we will show in Section 4, using polynomial functions yields lower fitting errors than using linear functions. Thus, our proposal leads to smaller index sizes and faster queries.

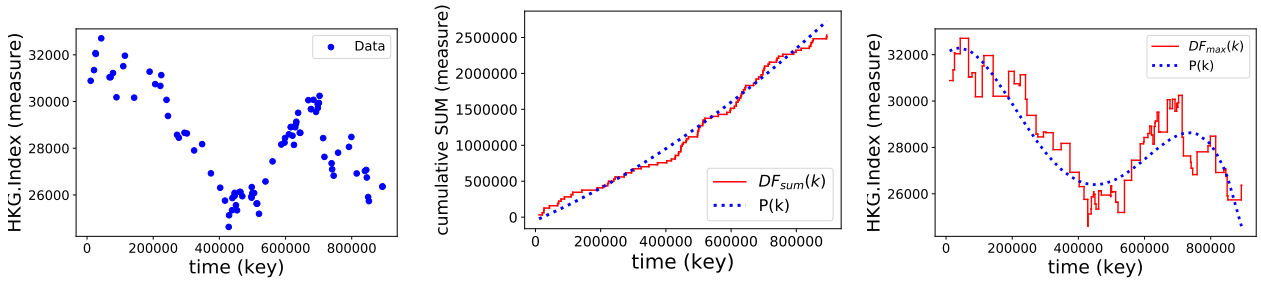
The key technical challenges are as follows. (1) How to find polynomial functions with low approximation error efficiently? (2) How to answer range aggregate queries with error guarantees? (3) How to support common aggregate functions (e.g., COUNT, SUM, MIN, MAX) and multi-dimensional data?

To tackle these challenges, we develop a polynomial-based indexing approach (PolyFit) for processing approximate range aggregate queries. Our contributions are summarized as follows.

- To the best of our knowledge, this is the first study that utilizes polynomial functions to learn indexes that support approximate range aggregate queries.
- PolyFit supports multiple types of range aggregate queries, including COUNT, SUM, MIN and MAX with guaranteed deterministic absolute and relative error bounds.
- Experiment results show that PolyFit achieves significant speedups, compared with the closest related works [28, 44], and traditional exact/approximate methods. For instance, for the OpenStreetMap dataset with 100M records, our index occupies only 4 MBytes and offers 5 μs query response time (per 2-dimensional range COUNT query).

The rest of the paper is organized as follows. We first review the related work in Section 2. Next, we introduce preliminaries in Section 3. Then, we present our index construction techniques in Section 4 and cover how to answer approximate range aggregate

*This work was supported by grant GRF 152050/19E from the Hong Kong RGC.



(a) timestamped index values (b) function for range SUM queries (c) function for range MAX queries

Figure 1: Stock market index values, 1-dimensional keys: discrete data points vs. continuous function

queries in Section 5. Next, we extend our proposal to datasets with two keys in Section 6. Lastly, we present experiments in Section 7 and conclude in Section 8.

2 RELATED WORK

Range aggregate queries are used frequently in analytics applications and constitute important functionality in OLAP and data warehousing [11, 12, 20, 23, 38, 40, 56, 69]. Exact solutions are based on prefix-sum arrays [38] or aggregate R-trees [59]. Due to the need for real-time performance in some applications (e.g., μs-level response time [74]), many proposals exist that aim to improve the efficiency of range aggregate queries. These proposals can be classified as being either data-driven or query-driven. In addition, we also review some other studies, including learned indexes, and time series databases, which are also related to this work.

Data-driven proposals build statistical models of a dataset for estimating query selectivity or the results of range aggregate queries. These models employ multi-dimensional histograms [39, 52, 57, 68], data sampling [9, 34, 36, 51, 60, 65], or kernel density estimation [32, 33, 37]. Although such proposals that compute approximate results are much faster than exact solutions, e.g., achieving ms (10^{-3}) level response time [61], they still do not offer real-time performance (e.g., μs level [74]). Furthermore, these proposals do not offer theoretical approximation error guarantees.

The **query-driven approaches** utilize query workloads to build statistical models of datasets. Typical methods include error-feedback histograms [8, 10, 49], max-entropy histograms [55, 64], and learning-based models [53, 66]. In addition, Park et al. [61] explore the approach of using mixture probabilistic models. These methods assume that new queries follow historical query workload distributions. However, as one study [13] observes, this assumption may not always hold in practice. Further, even when this assumption is valid, the number of queries that are similar to those used for training may be much smaller if the queries follow a power law distribution [73], which can cause poor accuracy and may render it impossible to obtain useful approximation error guarantees for range aggregate queries.

Recently, **learning-based methods** have been used to construct more compact and effective index structure, that hold potential to accelerate database operations. Kraska et al. [44] propose the RMI index, which incorporates different machine learning models, e.g., linear regression and deep-learning, to improve the efficiency of range queries. Galakatos et al. [28] develop the FITing-tree, which is a segment-tree-like structure [22, 71] that can significantly improve the efficiency of exact point queries. Ferragina et al. [27] further support efficient update operations for range queries. Wang et al. [70] extend this learning-based approach to the spatial domain with their learned Z-order model that aims to support fast spatial indexing. However, there are two main differences between these proposals and our proposal. First, they either support range queries [27, 44, 70] or point queries [28], but not range aggregate queries. Second, we are the first to exploit polynomial functions to build index structures for approximate range aggregate queries.

In the **time series database** community, some research studies utilize mathematical models to approximate time series data. Representative approaches include piecewise linear approximation [25, 41–43, 50], discrete wavelet transform [15, 62], discrete Fourier transform [26, 63], and their combinations [40, 58]. However, these studies focus on either time series similarity search (e.g., range or nearest neighbor queries) or time series compression and they are not designed to answer the range aggregate queries we target. Some of these studies also utilize piecewise linear approximation [25, 41, 42, 50] to approximate time-series, which we also do. In contrast, we achieve better performance by utilizing nonlinear (polynomial) functions to approximate curves, which can reduce the number of segments dramatically. Furthermore, we can also support the segmentation of surfaces (e.g., Figure 9(b)), rather than only 1-D curves.

3 PRELIMINARIES

First, we define range aggregate queries and their approximate versions in Section 3.1. Then, we discuss the baselines for answering exact range aggregate queries in Section 3.2. Table 1 summarizes frequently used symbols in this paper.

Table 1: Symbols

Symbol	Description
$\mathcal{D}$	dataset
n	number of records in $\mathcal{D}$
R_{count}	range COUNT query
R_{sum}	range SUM query
R_{min}	range MIN query
R_{max}	range MAX query
CF_{sum}	cumulative function for range SUM query
DF_{max}	key-measure function for range MAX query
$\mathbb{P}(k)$	polynomial function
I	interval
deg	degree of polynomial function

3.1 Problem Definition

We focus on the setting that a range aggregate query specifies a *key* attribute (for range selection) and a *measure* attribute for

aggregation. We shall consider the setting of two keys in Section 6. As such, the dataset $\mathcal{D}$ is a set of $(key, measure)$ records, i.e., $\mathcal{D} = \{(k_1, m_1), (k_2, m_2), ..., (k_n, m_n)\}$. For ease of discussion, we assume that key values are distinct and measure values are numerical. We leave the discussion of repeated keys and negative measure values in Appendices A.3 and A.4 [48]. Then we define a range aggregate query as follows.

Definition 3.1. Let $\mathcal{G}$ be an aggregate function (e.g., COUNT, SUM, MIN, MAX) on a measure attribute. Given a dataset $\mathcal{D}$ and a key range $[l_q, u_q]$, we define V as the following multi-set

$$V = \{m \mid (k, m) \in \mathcal{D} \wedge l_q \leq k \leq u_q\}$$

and then define the result of the range aggregate query as

$$R_{\mathcal{G}}(\mathcal{D}, [l_q, u_q]) = \mathcal{G}(V). \tag{1}$$

We aim to develop efficient methods for obtaining an approximate result of $R_{\mathcal{G}}(\mathcal{D}, [l_q, u_q])$ with two types of error guarantees [29, 30], namely the absolute error guarantee (cf. Problem 1) and the relative error guarantee (cf. Problem 2).

Problem 1 (Q_{abs}). *Given an absolute error ε_{abs} and a range aggregate query, we ask for an approximate result A_{abs} such that:*

$$|A_{abs} - R_{\mathcal{G}}(\mathcal{D}, [l_q, u_q])| \leq \varepsilon_{abs} \tag{2}$$

Problem 2 (Q_{rel}). *Given a relative error ε_{rel} and a range aggregate query, we ask for an approximate result A_{rel} such that:*

$$\left| \frac{A_{rel} - R_{\mathcal{G}}(\mathcal{D}, [l_q, u_q])}{R_{\mathcal{G}}(\mathcal{D}, [l_q, u_q])} \right| \leq \varepsilon_{rel} \tag{3}$$

3.2 Baselines: Exact Methods

We proceed to discuss exact methods for answering range SUM queries and range MAX queries. These methods can be easily extended to support COUNT and MIN, respectively.

3.2.1 **Exact method for range SUM queries**. First, we define the key cumulative function as $CF_{sum}(k)$:

$$CF_{sum}(k) = R_{sum}(\mathcal{D}, [-\infty, k]). \tag{4}$$

The additive property of CF_{sum} enables us to compute the exact result of the range SUM query as:

$$R_{sum}(\mathcal{D}, [l_q, u_q]) = CF_{sum}(u_q) - CF_{sum}(l_q). \tag{5}$$

Then, we discuss how to obtain the terms $CF_{sum}(l_q)$ and $CF_{sum}(u_q)$ efficiently. Although CF_{sum} is a continuous function, it can be expressed by a discrete data structure in finite space. Specifically, we presort dataset $\mathcal{D}$ in ascending key order and then follow this order to construct a key-cumulative array of entries $(k, CF_{sum}(k))$. At query time, the terms $CF_{sum}(l_q)$ and $CF_{sum}(u_q)$ are obtained by performing binary search on the above key-cumulative array. This step takes $O(\log n)$ time.

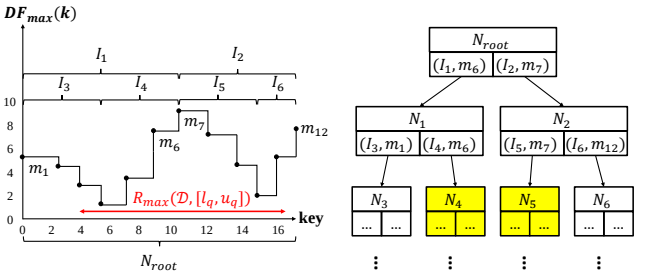
As a remark, this key-cumulative array is similar to the prefix-sum array [38]. The difference is that our array allows floating-point search keys, while the prefix-sum array does not.

3.2.2 **Exact method for range MAX queries**. First, we define the key-measure function $DF_{max}(k)$ in Equation 6 to capture the data distribution in the dataset $\mathcal{D}$. In the definition, we assume that each pair (k_i, m_i) in $\mathcal{D}$ is arranged in ascending order by the key.

$$DF_{max}(k) = \begin{cases} m_1 & \text{if } k_1 \leq k < k_2 \\ \vdots & \vdots \\ m_i & \text{if } k_i \leq k < k_{i+1} \\ \vdots & \vdots \\ m_n & \text{if } k = k_n \\ -\infty & \text{otherwise} \end{cases} \tag{6}$$

Figure 2(a) exemplifies the function $DF_{max}(k)$.

An aggregate max-tree [59] (cf. Figure 2(b)) can be built to answer range MAX queries. In this tree, each internal node stores two entries, where each entry stores an interval and the maximum measure within that interval (e.g., (I_1, m_6) and (I_2, m_7) are two entries of the root node N_{root}). We then explain how to process the query $R_{max}(\mathcal{D}, [l_q, u_q])$, whose query range is indicated by the red line in Figure 2(a). In Figure 2(b), we start from the root of the tree. If the interval of an entry intersects with the query range (e.g., I_1 and I_2 in Figure 2(a)), we visit its child nodes (e.g., N_1 and N_2). When the interval of an entry (e.g., I_4 and I_5 in Figure 2a) is covered by the query range, we directly use its stored aggregate value without visiting its child nodes (e.g., yellow nodes in Figure 2b). During the traversal, we keep track of the maximum measure seen so far. This procedure takes $O(\log n)$ time as we check at most two branches per level.



(a) (key, measure)-pairs, red line denotes the max query

(b) Hierarchical max-tree, yellow nodes are skipped during traversal

Figure 2: Aggregate MAX tree

4 INDEX CONSTRUCTION

Traditional index structures (e.g., B-tree [21]) need to store n keys, where n is the cardinality of the dataset $\mathcal{D}$. Thus, the index size grows linearly with the data size. To reduce the index size dramatically, we plan to index a limited number of functions (instead of n keys).

As a case study, we compare existing fitting functions [28, 44] with our fitting function (polynomial) on a real dataset (the Hong Kong 40 Index in 2018 [5]) in Figure 3. The exact key-measure function $DF_{max}(k)$ exhibits a complex shape. Observe that linear functions, e.g., linear regression $LR(k)$ [44] and linear segment $FIT(k)$ [28], cannot accurately approximate the exact function. In this paper, we adopt the polynomial function $\mathbb{P}(k)$, which captures the nonlinear property[1] and achieves a better approximation of $DF_{\max}(k)$. In this example, $\mathbb{P}(k)$ is a degree-4 polynomial function (blue dotted line).

We introduce our indexing framework in Figure 4. First, we convert the dataset into the following exact function $F(k)$ based

[1] As a remark, other types of nonlinear functions (e.g., logarithmic and trigonometric functions) require higher computation cost than polynomial functions. Thus, we leave other types of nonlinear functions as future work.

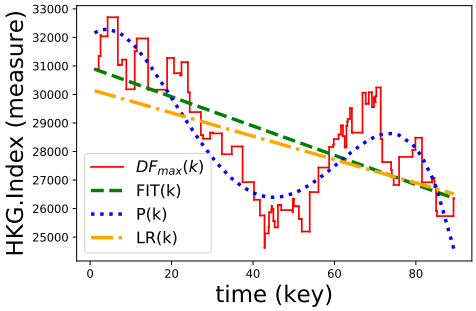


Figure 3: Curve fitting of the HKG 40 Index in 2018 [5]

on the aggregate function $\mathcal{G}$ and the functions in Section 3.2.

$$F(k) = \begin{cases} CF_{sum}(k) & \text{if } \mathcal{G} = \texttt{SUM} \\ DF_{max}(k) & \text{if } \mathcal{G} = \texttt{MAX} \end{cases} \tag{7}$$

We plan to compute an error-bounded approximation of $F(k)$ by using a sequence of polynomial functions. In Section 4.1, we examine how to find the best polynomial fitting of $F(k)$ in a given key interval I. Then, in Section 4.2, we propose a segmentation method for $F(k)$ in order to minimize the index size subject to a given deviation threshold. Finally, in Section 4.3, we discuss how to build an index for a sequence of polynomial functions.

4.1 Polynomial Fitting in a Key Interval

We discuss how to find the best fitting polynomial function of $F(k)$ in a given key interval I. First, we express a polynomial function $\mathbb{P}(k)$ as follows:

$$\mathbb{P}(k) = \sum_{j=0}^{deg} a_j k^j, \tag{8}$$

where deg is the degree and each a_j is a coefficient. Note that the choice of deg entails tradeoffs between the fitting error and the online query evaluation cost. We discuss the choice of deg in Section 5.3.

We formulate the following optimization problem in order to minimize the fitting error between $\mathbb{P}(k)$ and $F(k)$.

Definition 4.1. Let $F(k)$ be the exact function and I be a given key interval. Let $k_1, k_2, \cdots, k_\ell$ be the keys of $\mathcal{D}$ in interval I. We aim to find polynomial coefficients, $a_0, a_1, \cdots, a_{deg}$ that minimize the following error:

$$E(I) = \min_{a_0, a_1, \ldots, a_{deg} \in \mathbb{R}} \max_{1 \le i \le \ell} |F(k_i) - \mathbb{P}(k_i)| \tag{9}$$

This is equivalent to the following linear programming problem, where the coefficients $a_0, a_1, \cdots, a_{deg}$ and t are variables.

$$\begin{cases} \text{MINIMIZE} \quad t \\ \text{SUBJECT TO:} \\ -t \le F(k_1) - (a_{deg} k_1^{deg} + \ldots + a_2 k_1^2 + a_1 k_1 + a_0) \le t \\ -t \le F(k_2) - (a_{deg} k_2^{deg} + \ldots + a_2 k_2^2 + a_1 k_2 + a_0) \le t \\ \ldots \\ -t \le F(k_\ell) - (a_{deg} k_\ell^{deg} + \ldots + a_2 k_\ell^2 + a_1 k_\ell + a_0) \le t \\ \forall a_i \in \mathbb{R} \end{cases} \tag{10}$$

It takes $O(\ell^{2.5})$ time to solve the above linear programming problem (Equation 10) [46]. In our experimental study, we adopt the IBM CPLEX linear programming library as the LP Solver, which is believed to be the most reliable and efficient among other implementations [31]. We discuss some subtle issues like precision limitations in Section 5.3.

4.2 Minimal Index Size with Bounded Error

To support approximate query evaluation (in Section 5), we require that the fitting polynomial functions should satisfy a given error constraint. However, a single polynomial function is unlikely to fit accurately for the entire key domain. Thus, we propose to partition the key domain into intervals $I_1, I_2, \cdots, I_h$ so that each interval I_i satisfies the following requirement:

$$E(I) \le \delta,$$

where δ is a given deviation threshold. For instance, in Figure 5, the key domain is partitioned into two intervals I_1 and I_2 so that the best fitting polynomial function in each interval satisfies the error requirement.

To achieve a small index size, we aim to minimize the number of intervals (i.e., h in Figure 4). An existing dynamic programming (DP) approach [47], though designed for piecewise linear functions, can be adapted to solve our partitioning problem of $F(k)$. However, this method takes $O(n^2 \times \ell_{max}^{2.5})$ time[2], where ℓ_{max} is the maximum number of keys covered by any interval. Obviously, this method does not scale well with the data size n.

In Section 4.2.1, we present a more efficient method, called greedy segmentation (GS), to segment the exact function $F(k)$. As we show later, the time complexity of GS is $O(n \times \ell_{max}^{2.5})$, which scales well with the data size n. Then, in Section 4.2.2, we show that GS is guaranteed to return the optimal solution.

4.2.1 **Greedy Segmentation (GS) Method.** We present the pseudo-code of the Greedy Segmentation (GS) method in Algorithm 1. It examines the key domain from left to right (line 2). In each iteration, it expands the interval I by including the next key (line 3), calls an LP solver on the interval I to obtain a fitting function $\mathbb{P}_{now}$ (line 4), and tests whether it fulfills the error requirement. When this test fails (i.e., $E(I) > \delta$), we conclude that the previous interval is a maximal interval and thus insert its corresponding fitting function $\mathbb{P}_{prev}$ into the result. The above procedure is repeated until all keys are covered.

Algorithm 1 Greedy Segmentation (GS)

Input: function $F(k)$, degree deg, deviation threshold δ
Output: sequence of polynomial functions $\text{Seq}_{\mathbb{P}}$

```
1: Seq_P ← ∅;  l ← 1;  P_prev ← null
2: for u ← 2 to n do
3:     I ← [k_l, k_u]                    ▷ the interval for polynomial function P
4:     P_now ← call LP solver on I       ▷ Equation 10
5:     if E(I) > δ or u = n then         ▷ Equation 9
6:         insert P_prev into Seq_P
7:         l ← u
8:     P_prev ← P_now
9: return Seq_P
```

The time complexity of GS is $O(n\ell_{max}^{2.5})$ because it invokes $O(n)$ calls to the LP solver, where each call takes $O(\ell_{max}^{2.5})$ time [46]. We further accelerate GS by applying an existing exponential search technique [14], which can reduce the number of LP calls per interval by $\frac{\ell}{\log \ell}$ times. With this technique, GS takes only 70 seconds (cf. Section 7.2.2) to complete for a real dataset with 1 million data points. This is acceptable for many data analytics tasks (with static datasets) in OLAP. In our experiments, we find that ℓ_{max} usually ranges between hundreds and thousands, thus

[2]Recall that the state-of-the-art linear programming solver [46] takes $O(\ell_{max}^{2.5})$ time for each curve-fitting problem (cf. Equation 10).

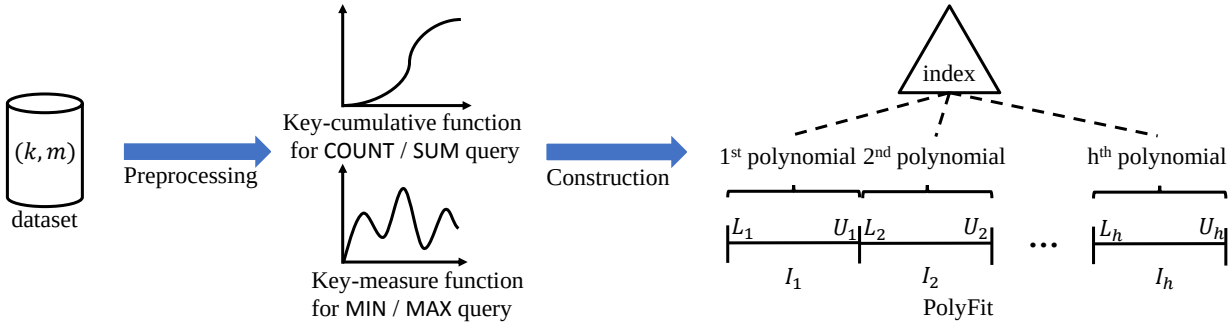


Figure 4: Indexing framework for PolyFit, each leaf entry stores a polynomial function

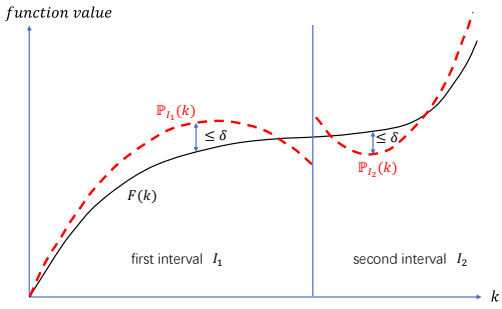


Figure 5: Fitting $F(k)$ with multiple polynomial functions, subject to the deviation threshold δ

the term $O(\ell_{max}^{2.5})$ is acceptable in practice. In Appendix [48], we discuss how to utilize parallel computation to further improve the construction time.

4.2.2 **GS is Optimal**. We first prove the following property (Lemma 4.2) of our curve fitting problem (cf. Definition 4.1).

Lemma 4.2. *Let I_l and I_u be two intervals, which contain two sets of keys S_l and S_u, respectively. If $S_l \subseteq S_u$, then $E(I_l) \leq E(I_u)$.*

Proof. Recall that the value of $E(I)$ (cf. Equation 9) is equal to the minimum value of the optimization problem (Equation 10). Since S_l is a subset of S_u, the set of constraints for solving $E(I_l)$ is also the subset of constraints for solving $E(I_u)$. Thus, for the minimization problem in Equation 10, the possible solution space for S_l is a superset of the possible solution space for S_u. Therefore, we conclude that $E(I_l) \leq E(I_u)$. □

Based on Lemma 4.2, we then show that GS produces the fewest polynomial functions (cf. Theorem 4.3), i.e., the optimal solution.

Theorem 4.3. *GS always produces the optimal number of functions (with respect to the given parameters deg and δ).*

Proof. We denote the minimum key and the maximum key of an interval I by $I.\min$ and $I.\max$, respectively.

Let $\mathcal{I}^*_{\text{OPT}} = (I^{(1)}_{\text{OPT}}, I^{(2)}_{\text{OPT}}, \cdots)$ and $\mathcal{I}^*_{\text{GS}} = (I^{(1)}_{\text{GS}}, I^{(2)}_{\text{GS}}, \cdots)$ be two ascending sequences of intervals for the optimal solution and our GS method, respectively (i.e., $I^{(i)}.\max < I^{(i+1)}.\min$ for $i = 1, 2, \ldots, n-1$). Every interval I in $\mathcal{I}^*_{\text{OPT}}$ and $\mathcal{I}^*_{\text{GS}}$ must satisfy $E(I) \leq \delta$. We now prove the theorem by mathematical induction.

In the base step, we consider the first interval in each sequence. Since both GS and OPT must cover the key domain, we have:

$$I^{(1)}_{\text{GS}}.\min = I^{(1)}_{\text{OPT}}.\min$$

According to GS, the first interval $I^{(1)}_{\text{GS}}$ is maximal, because a longer interval would violate the deviation threshold δ. Thus, we have:

$$I^{(1)}_{\text{GS}}.\max \geq I^{(1)}_{\text{OPT}}.\max \tag{11}$$

In the inductive step, assume that the first ℓ intervals of the two sequences satisfy the following property:

$$I^{(\ell)}_{\text{GS}}.\max \geq I^{(\ell)}_{\text{OPT}}.\max \tag{12}$$

Since $\mathcal{I}^*_{\text{OPT}}$ and $\mathcal{I}^*_{\text{GS}}$ are ascending sequences of intervals, Equation 12 implies the following:

$$I^{(\ell+1)}_{\text{GS}}.\min \geq I^{(\ell+1)}_{\text{OPT}}.\min \tag{13}$$

Now, we consider two cases for comparing $I^{(\ell+1)}_{\text{GS}}$ and $I^{(\ell+1)}_{\text{OPT}}$.

Case 1:

$$I^{(\ell+1)}_{\text{GS}}.\max \geq I^{(\ell+1)}_{\text{OPT}}.\max$$

In this case, the first $\ell+1$ intervals of GS cover all keys in the first $\ell+1$ intervals of OPT.

Case 2:

$$I^{(\ell+1)}_{\text{GS}}.\max < I^{(\ell+1)}_{\text{OPT}}.\max \tag{14}$$

Consider the interval $I' = [I^{(\ell+1)}_{\text{GS}}.\min, I^{(\ell+1)}_{\text{OPT}}.\max]$. By using Equations 13 and 14, we obtain: $I' \subset I^{(\ell+1)}_{\text{OPT}}$. By Lemma 4.2, we get: $E(I') \leq E(I^{(\ell+1)}_{\text{OPT}})$. Since $E(I^{(\ell+1)}_{\text{OPT}}) \leq \delta$, we get: $E(I') \leq \delta$.

Observe that I' has the same minimum key as $I^{(\ell+1)}_{\text{GS}}$ but a larger maximum key than $I^{(\ell+1)}_{\text{GS}}$. Since I' does not pass the error test in GS, we get $E(I') > \delta$. This contradicts the statement $E(I') \leq \delta$.

Therefore, only the first case is true, and we have:

$$I^{(\ell+1)}_{\text{GS}}.\max \geq I^{(\ell+1)}_{\text{OPT}}.\max$$

This means GS always covers no fewer keys than OPT with the same number of intervals. Thus, GS produces the optimal number of functions. □

4.3 Indexing of polynomial functions

In our experimental study, the number of intervals (for polynomial functions) ranges from 100 to 1000. We adopt existing index structures on these intervals to support fast query evaluation. Specifically, we employ an in-memory index called the STX B-tree [6] to index intervals. In each internal node entry, we maintain an additional attribute to store the aggregate value of its subtree. In each leaf node entry, we store an interval and its corresponding polynomial model (in the form of coefficients). In summary, this index is similar to the aggregate tree exemplified in Figure 2(b), except that we store polynomial models in leaf nodes.

5 APPROXIMATE QUERY EVALUATION

We present our framework for answering approximate range aggregate queries in Figure 6. The first step is to compute an initial approximate result quickly by using our index (PolyFit). Then, we check whether the error condition is satisfied and refine the approximate result if necessary. We discuss how to answer

the approximate range SUM query and the approximate range MAX query in Sections 5.1 and 5.2, respectively. Finally, in Section 5.3, we discuss how to tune our index parameters (e.g., deg, δ) in order to optimize the query response time.

5.1 Approximate range SUM Query

Given the query range $[l_q, u_q]$, we propose to compute the approximate result as:

$$\tilde{A}_{sum} = \mathbb{P}_{I_u}(u_q) - \mathbb{P}_{I_l}(l_q), \tag{15}$$

where I_l and I_u denote the intervals of $\mathbb{P}$ that contain the values l_q and u_q, respectively.

Then, we show the error conditions for Q_{abs} (cf. Problem 1) and Q_{rel} (cf. Problem 2).

Error condition for Q_{abs}

Given the absolute error ε_{abs}, we recommend to use the deviation threshold $\delta = \frac{\varepsilon_{abs}}{2}$ in constructing PolyFit. With this setting, the following lemma offers the absolute error guarantee for the approximate result $\tilde{A}_{sum}$ (in Equation 15).

Lemma 5.1. *If $\delta = \frac{\varepsilon_{abs}}{2}$, then $\tilde{A}_{sum}$ (in Equation 15) satisfies the absolute error guarantee with respect to ε_{abs}.*

Proof. Let I_l and I_u be two intervals (in PolyFit) which contain l_q and u_q (of the query range $[l_q, u_q]$), respectively. Based on the deviation threshold guarantee in Section 4.2.2, we obtain:

$$|CF_{sum}(l_q) - \mathbb{P}_{I_l}(l_q)| \leq \delta$$

$$|CF_{sum}(u_q) - \mathbb{P}_{I_u}(u_q)| \leq \delta$$

By combining them, we have:

$$CF_{sum}(u_q) - CF_{sum}(l_q) - 2\delta \leq \tilde{A}_{sum} \leq CF_{sum}(u_q) - CF_{sum}(l_q) + 2\delta$$

By using Equation 5, we have:

$$R_{sum}(\mathcal{D}, [l_q, u_q]) - 2\delta \leq \tilde{A}_{sum} \leq R_{sum}(\mathcal{D}, [l_q, u_q]) + 2\delta$$

Since $\delta = \frac{\varepsilon_{abs}}{2}$, $\tilde{A}_{sum}$ satisfies the absolute error guarantee ε_{abs}. □

Error condition for Q_{rel}

In this scenario, there is no specific preference for setting the deviation threshold δ when constructing PolyFit. The following lemma suggests a condition to test whether $\tilde{A}_{sum}$ satisfies the relative error guarantee. If this test fails, we resort to the exact method (cf. Section 3.2.1) to obtain the exact result.

Lemma 5.2. *If $\tilde{A}_{sum} \geq 2\delta(1 + \frac{1}{\varepsilon_{rel}})$, then $\tilde{A}_{sum}$ (in Equation 15) satisfies the relative error guarantee with respect to ε_{rel}.*

Proof. Like in the proof of Lemma 5.1, we can derive Equations 16 and 17.

$$|\tilde{A}_{sum} - R_{sum}(\mathcal{D}, [l_q, u_q])| \leq 2\delta \tag{16}$$

which also implies (by simple derivations):

$$R_{sum}(\mathcal{D}, [l_q, u_q]) \geq \tilde{A}_{sum} - 2\delta \tag{17}$$

Since δ and ε_{rel} must be positive, the given condition $\tilde{A}_{sum} \geq 2\delta(1 + \frac{1}{\varepsilon_{rel}})$ implies that $\tilde{A}_{sum} > 2\delta$ and $\frac{2\delta}{\tilde{A}_{sum} - 2\delta} \leq \varepsilon_{rel}$.

Dividing Equation 16 by Equation 17, we obtain the following inequality (under the condition $\tilde{A}_{sum} > 2\delta$).

$$\frac{|\tilde{A}_{sum} - R_{sum}(\mathcal{D}, [l_q, u_q])|}{R_{sum}(\mathcal{D}, [l_q, u_q])} \leq \frac{2\delta}{\tilde{A}_{sum} - 2\delta}$$

This completes the proof because $\frac{2\delta}{\tilde{A}_{sum} - 2\delta} \leq \varepsilon_{rel}$. □

The overall query algorithm

We summarize the query algorithm for both types of error guarantees in Algorithm 2. The processing for Q_{abs} is composed of two parts: index search T_1 (i.e., Lines 1-2) and function evaluation T_2 (i.e., Line 3). The processing for Q_{rel} includes T_1, T_2, and possible refinement T_3 (i.e., Lines 4-6). The time complexity of T_1, T_2, and T_3 are $O(\log(|\text{Seq}_{\mathbb{P}}|))$, $O(deg)$, and $O(\log|\mathcal{D}|)$ respectively.

Algorithm 2 Query Processing for SUM (or COUNT)

Input: $\text{Seq}_{\mathbb{P}}$ (output from Algorithm 1), l_q, u_q, $\mathcal{D}$, δ, Q_{type}
Output: Approximate query result A

1: $\mathbb{P}_{I_l} \leftarrow$ index search $\mathbb{P}$ from $\text{Seq}_{\mathbb{P}}$ that includes l_q
2: $\mathbb{P}_{I_u} \leftarrow$ index search $\mathbb{P}$ from $\text{Seq}_{\mathbb{P}}$ that includes u_q
3: $\tilde{A}_{sum} \leftarrow \mathbb{P}_{I_u}(u_q) - \mathbb{P}_{I_l}(l_q)$
4: **if** $Q_{type} = Q_{rel}$ **then**
5: **if** $\tilde{A}_{sum}$ fails the error condition of Lemma 5.2 **then**
6: $\tilde{A}_{sum} \leftarrow$ perform refinement on $\mathcal{D}$ ▷ Section 3.2.1
7: **return** $\tilde{A}_{sum}$

5.2 Approximate range MAX Query

The query method described in Section 3.2.2 can be applied here, except that we employ the index described in Section 4.3.

Given the query range $[l_q, u_q]$, we propose to compute the approximate result as:

$$\tilde{A}_{max} = \max\{\max_{k \in I_l, k \geq l_q} \mathbb{P}_{I_l}(k), \max_{k \in I_u, k \leq u_q} \mathbb{P}_{I_u}(k), \max_{N_j.I \subseteq [l_q, u_q]} N_j.max\} \tag{18}$$

where N_j denotes an internal node of the index built on top of $\text{Seq}_{\mathbb{P}}$. I_l and I_u denote the intervals of $\mathbb{P}$ that contain the values l_q and u_q, respectively.

The error conditions for Q_{abs} and Q_{rel} are presented in Lemmas 5.3 and 5.4 respectively. We omit their proofs; they are similar to the proofs of Lemmas 5.1 and 5.2.

Lemma 5.3. *If $\delta = \varepsilon_{abs}$, then $\tilde{A}_{max}$ (in Equation 18) satisfies the absolute error guarantee ε_{abs}.*

Lemma 5.4. *If $\tilde{A}_{max} \geq \delta(1 + \frac{1}{\varepsilon_{rel}})$, then $\tilde{A}_{max}$ (in Equation 18) satisfies the relative error guarantee ε_{rel}.*

We now discuss how to evaluate Equation 18 in greater detail. The third term is contributed by the inner nodes of the aggregate R-tree whose intervals are covered by $[l_q, u_q]$. Regarding the first two terms, it suffices to find the maximum values for $\mathbb{P}_{I_l}(k)$ and $\mathbb{P}_{I_u}(k)$ in regions $[l_q, U_{I_l}]$ and $[L_{I_u}, u_q]$, as shown in Figure 7, where U_{I_l} (L_{I_u}) is the upper (lower) end of the leaf node interval that l_q (u_q) overlaps. These values (i.e., red dots) can be calculated by checking the border points and the zero derivative points.

The overall query algorithm

We conclude the query algorithm for both types of error guarantees in Algorithm 3. The processing for Q_{abs} consists of two parts: index search T_1 (i.e., Line 3) and function evaluation T_2 (Lines 8-9). The processing for Q_{rel} includes T_1, T_2, and possible refinement T_3 (i.e., Lines 10-12). The time complexities of T_1 and T_3 are still $O(\log(|\text{Seq}_{\mathbb{P}}|))$ and $O(\log|\mathcal{D}|)$. However, for T_2, this includes calculating the zero derivative points within the intersection region. If the degree is between 1 and 5, closed-form equations exist, where the number of arithmetic operations in these cases are summarized in Table 2. Starting from degree 6, there is no closed-form equations, and thus require expensive

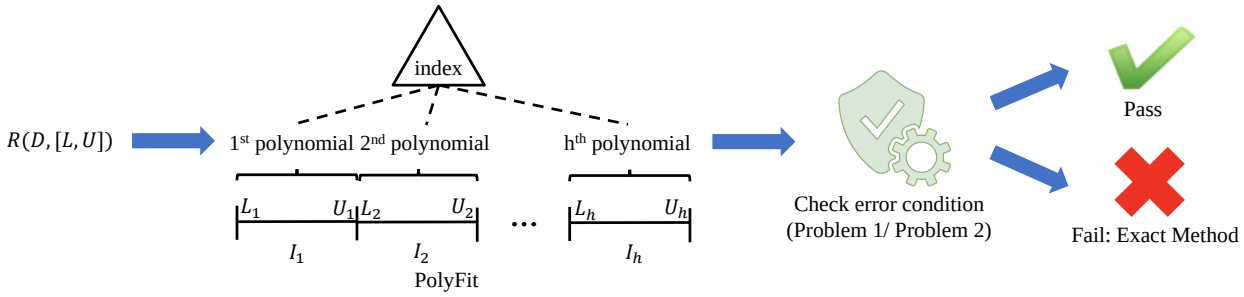


Figure 6: Querying framework for PolyFit

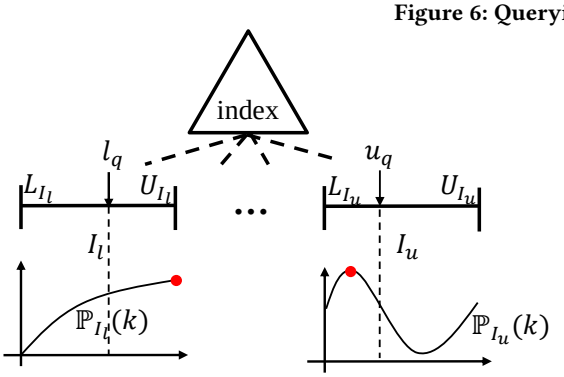


Figure 7: The maximum measure values (red dots) for two leaf nodes, which include l_q and u_q

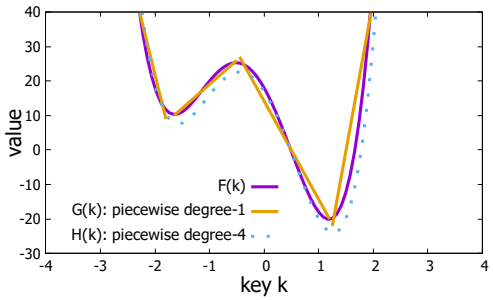


Figure 8: An example of degree selection

numerical evaluation methods like gradient descent [67]. In practice, we recommend to use degrees up to 3 for the approximate range MAX query.

Algorithm 3 Query Processing for MAX (or MIN)

Input: Aggregate R-tree N on $\text{Seq}_{\mathbb{P}}$, l_q, u_q, $\mathcal{D}$, δ, Q_{type}
Output: Approximate query result A

```
1:  Ã_max ← −∞
2:  if N is an internal node then
3:      update Ã_max based on aggregate R-tree's mechanism
4:  else
5:      for leaf element ℙ in N do
6:          if ℙ.I ∩ [l_q, u_q] ≠ ∅ then          ▷ the interval ℙ covered
7:              I* ← ℙ.I ∩ [l_q, u_q]
8:              β ← {x ∈ I* | ℙ'(x) = 0}          ▷ zero derivative points
9:              Ã_max ← max(Ã_max, max_{x∈β} ℙ(x), ℙ(I*.l), ℙ(I*.u)))
10: if N is root node and Q_type = Q_rel then
11:     if Ã_max fails the error condition of Lemma 5.4 then
12:         Ã_max ← perform refinement on 𝒟          ▷ Section 3.2.2
13: return A
```

Table 2: Number of arithmetic operations for calculating zero derivative points

degree	1	2	3	4	5
operations	0	2	up to 18	up to 261	up to 1612

5.3 Tuning *deg* and δ

We discuss the effect of our index parameters (i.e., *deg*, δ) on the query response time and examine how to tune them.

***How to tune the degree** deg?*

The exact function $F(k)$ is approximated by different polynomial functions with different degrees. For instance, in Figure 8, the exact function $F(k)$ is approximated, among others, by the following functions (within the deviation threshold δ): (i) a piecewise function $G(k)$ with four pieces of degree-1 functions, or (ii) a single-piece function $H(k)$ of degree-4. Based on our experimental findings (cf. Section 7.2.1), we recommend to set the degree to 2 or 3. In general, one could generate a random workload of queries to measure the performance of an index, and then test the performance of index structures using different degrees (e.g., from 1 to 4).

As a remark, it is not practical to use large degree, due to the limited precision of numeric data types in both the linear programming solver and the programming language [1, 2]. For example, IBM CPLEX uses κ (kappa) as a statistical measurement of numerical difficulties. In our experiments, the κ value of a degree-4 polynomial (1E+10) is much higher than that of a degree-1 polynomial (1E+05).

***How to tune** δ?*

The tuning of δ depends on the most frequent query type used in the given application. For Q_{abs} (i.e., Problem 1), if all users share the same absolute error threshold ε_{abs}, then it is used to derive the value of δ, according to Lemmas 5.1 and 5.3. Otherwise, we can select the value of δ such that it satisfies the error requirements for the majority of users (e.g., 80%).

For Q_{rel} (i.e., Problem 2), the processing includes three phases: index search, function evaluation, and refinement (cf. Algorithms 2 and 3). A large δ leads to fast index search but high refinement probability. In contrast, a small δ leads to slow index search but low refinement probability. Observe that refinement is often more expensive than index search. We recommend to pick a small δ such that most users avoid the refinement phase. In our experiments, we examine different values of δ (e.g., 25, 50, 100, 200, 500, and 1000) to identify the best setting in terms of the query response time.

6 EXTENSIONS: QUERIES WITH TWO KEYS

Previous sections consider range aggregate queries with a single key (cf. Definition 3.1). We now discuss how to support range aggregate queries with two keys (cf. Definition 6.1). Due to the space limit, we only consider the COUNT query. In Appendix A.5 [48], we discuss the case of more than two keys.

Definition 6.1. Let $\mathcal{D}$ be a set of records (u, v, w), where u, v, and w are the first key, the second key, and the measure, respectively. Given the query ranges $[l_q^{(1)}, u_q^{(1)}]$ and $[l_q^{(2)}, u_q^{(2)}]$ for u and v, respectively, we define the COUNT query as:

$$R_{count}(\mathcal{D}, [l_q^{(1)}, u_q^{(1)}][l_q^{(2)}, u_q^{(2)}]) = \text{COUNT}(V) \tag{19}$$

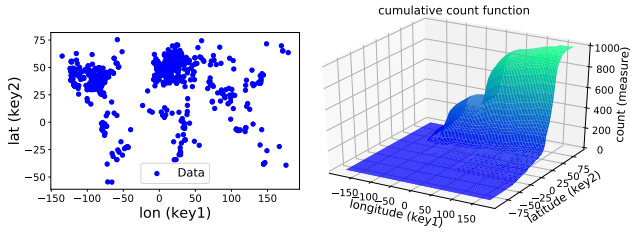
where V is the multi-set of measure values defined below:

$$V = \{m : (k^{(1)}, k^{(2)}, m) \in \mathcal{D}, l_q^{(1)} \leq k^{(1)} \leq u_q^{(1)}, l_q^{(2)} \leq k^{(2)} \leq u_q^{(2)}\}$$

We build the following key-cumulative function to represent the surface (cf. Figure 9), which is formulated in Definition 6.2.

Definition 6.2. The key-cumulative function with two keys for COUNT query is defined as $CF_{count}(u, v)$, where:

$$CF_{count}(u, v) = R_{count}(\mathcal{D}[-\infty, u][-\infty, v]) \tag{20}$$



(a) tweet locations as data points (b) function for range COUNT queries

Figure 9: Tweet locations, 2-dimensional keys: discrete data points vs. continuous function

The following equation enables us to answer the COUNT query quickly.

$$R_{count}(\mathcal{D}[l_q^{(1)}, u_q^{(1)}][l_q^{(2)}, u_q^{(2)}]) = CF_{count}(u_q^{(1)}, u_q^{(2)}) - CF_{count}(l_q^{(1)}, u_q^{(2)}) - CF_{count}(u_q^{(1)}, l_q^{(2)}) + CF_{count}(l_q^{(1)}, l_q^{(2)})$$

Then, we follow an idea similar to that used in Section 4.1 and utilize the polynomial surface $\mathbb{P}(u, v)$ to approximate the key cumulative function $CF_{count}(u, v)$ with two keys, where:

$$\mathbb{P}(u, v) = \sum_{i=0}^{deg} \sum_{j=0}^{deg} a_{ij} u^i v^j$$

By replacing $F(k_i)$ and $\mathbb{P}(k_i)$ in Equation 9 with $F(u_i, v_i)$ and $\mathbb{P}(u_i, v_i)$, respectively, we obtain a similar linear programming problem for obtaining the best parameters a_{ij}. However, unlike the one-dimensional case, it takes at least $O(n^2)$ to obtain the minimum number of segmentations when using the GS method (cf. Section 4.2.1), which is infeasible even for small-scale datasets (e.g., 10000 points). Instead, we propose a heuristics-based solution that performs quad-tree-like segmentations. As illustrated in Figure 10, when a region does not fulfill the error guarantee δ (e.g., white rectangles), it is decomposed into four smaller regions. This procedure terminates when all regions satisfy the error guarantee δ.

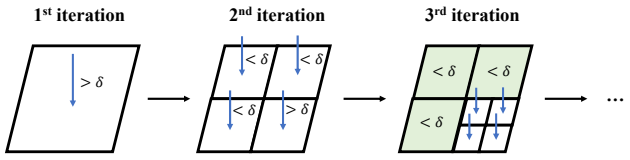


Figure 10: Quad-tree based approach for obtaining the segmentation

After building the PolyFit index structure, we utilize a similar approach in Section 5 to answer range aggregate queries with theoretical guarantees (cf. Lemmas 6.3 and 6.4).

Given the query range $[l_q^{(1)}, u_q^{(1)}]$ for u and $[l_q^{(2)}, u_q^{(2)}]$ for v, we propose to compute the approximate result as:

$$\tilde{A}_{count} = \mathbb{P}_{I_{uu}}(u_q^{(1)}, u_q^{(2)}) - \mathbb{P}_{I_{lu}}(l_q^{(1)}, u_q^{(2)}) - \mathbb{P}_{I_{ul}}(u_q^{(1)}, l_q^{(2)}) + \mathbb{P}_{I_{ll}}(l_q^{(1)}, l_q^{(2)}) \tag{21}$$

where I_{uu}, I_{lu}, I_{ul}, and I_{ll} denote the coverage regions of $\mathbb{P}$ that $(u_q^{(1)}, u_q^{(2)})$, $(l_q^{(1)}, u_q^{(2)})$, $(u_q^{(1)}, l_q^{(2)})$, and $(l_q^{(1)}, l_q^{(2)})$ overlap, respectively. These regions could be efficiently found with the same quad-tree index used in construction.

LEMMA 6.3. *If we set* $\delta = \frac{\varepsilon_{abs}}{4}$, *then* $\tilde{A}_{count}$ *satisfies the absolute error guarantee* ε_{abs}.

LEMMA 6.4. *If* $\tilde{A}_{count} \geq 4\delta(1 + \frac{1}{\varepsilon_{rel}})$, *then* $\tilde{A}_{count}$ *satisfies the relative error guarantee* ε_{rel}.

The proofs of Lemma 6.3 and 6.4 are similar to those of Lemmas 5.1 and 5.2, respectively.

7 EXPERIMENTAL EVALUATION

We introduce the experimental setting in Section 7.1. Then, we investigate the performance of PolyFit in Section 7.2. Next, we compare PolyFit and error-bounded competitors on real datasets in Section 7.3. After that, we compare the response time of PolyFit with other heuristic methods in Section 7.4. Lastly, we compare the construction times of all methods in Section 7.5.

7.1 Experimental Setting

We use three real large-scale datasets (0.9M to 100M records) to evaluate the performance. They are summarized in Table 3. For each dataset, we randomly generate 1000 queries. In the single-key case, we randomly choose two key values in the datasets as the start and end points of each query interval. In the two-key case, we randomly sample rectangles from the dataset as query regions. In our experiments, we focus on COUNT and MAX queries. Nevertheless, our methods are readily applicable to SUM and MIN queries.

Table 3: Datasets

Name	Size	Key(s)	Measure	Aggregate function
HKI [5]	0.9M	timestamp	index value	MAX
TWEET [19]	1M	latitude	# of tweets	COUNT
OSM [7]	100M	latitude, longitude	# of records	COUNT

Table 4 summarizes different methods for supporting range aggregate queries. We classify these methods based on five features: (i) whether it provides absolute error guarantees (cf. Problem 1 (Q_{abs})), (ii) whether it provides relative error guarantees (cf. Problem 2 (Q_{rel})), (iii) whether it supports queries with two keys (cf. Section 6), (iv) whether it supports the COUNT query, and (v) whether it supports the MAX query.

We first introduce the methods that can satisfy deterministic error guarantees (i.e., those with ✓ or △ in the Q_{abs} and Q_{rel} columns in Table 4). The aR-tree [59] is a traditional tree-based method for answering exact COUNT and MAX queries. The MRTree [45] extends the aR-tree by utilizing progressive lower and upper bounds to answer approximate COUNT and MAX queries with error guarantees. In addition, both the aR-tree and the MRTree can support the range aggregate queries with two keys. With simple modifications, the learned-index methods, including RMI [44], FITing-tree [28], and PGM [27], can be extended to support range aggregate queries with both absolute and relative error guarantees. However, they are unable to support queries with two keys and the MAX query. Due to the space limitation, we cover the modifications and parameter tuning in our technical report (cf. Appendix in [48]). PolyFit supports all these five features. By

Table 4: Methods for range aggregate queries

✓ Directly support △ Extend to support × Cannot support

Method	Q_{abs}	Q_{rel}	2 keys	COUNT	MAX
aR-tree [59]	✓	✓	✓	✓	✓
MRTree [45]	✓	✓	✓	✓	✓
RMI [44]	△	△	×	✓	×
FITing-tree [28]	△	△	×	✓	×
PGM [27]	△	△	×	✓	×
PolyFit (ours)	✓	✓	✓	✓	✓
Hist [68]	×	×	×	✓	×
S-tree [6]	×	×	×	✓	×
S2 [35]	×	×	✓	✓	×
VerdictDB [60]	×	×	✓	✓	×
DBest [53]	×	×	✓	✓	×
PLATO [50]	×	×	×	✓	×

default, we follow Lemmas 5.1, 5.3, and 6.3 to set the δ values in Problem 1 (Q_{abs}), for different absolute error threshold ε_{abs}. In addition, we adopt $\delta = 100$ in PolyFit for the experiments with two keys in Problem 2 (Q_{rel}).

We then discuss the methods that are unable to fulifll the deterministic error guarantee (i.e., the methods with × in the Q_{abs} and Q_{rel} columns in Table 4). Hist [68] adopts the entropy-based histogram for answering the COUNT query. The S-tree prebuilds the STX B-tree [6] on top of a sampled subset of each dataset. S2 [35] and VerdictDB [60] are sampling-based approaches that can only provide probabilistic error guarantees. By default, we set the probability to 0.9 in our experiments. Both DBest [53] and PLATO [50] are the state-of-the-art methods in approximate query processing and time series databases, respectively, that can be also adapted to answer approximate range aggregate queries. Since these methods cannot provide deterministic error guarantees, we regard them as heuristic methods.

We implemented all methods in C++ and conducted experiments on an Intel Core i7-8700 3.2GHz PC using WSL (Windows 10 Subsystem for Linux).

7.2 PolyFit Tuning

In this section, we investigate two research questions for PolyFit, namely (1) how does the degree *deg* affect the query response time of PolyFit? (2) how does the degree *deg* affect the construction time of PolyFit?

7.2.1 **Effect of *deg* on the query response time**. Recall that we need to select the degree *deg* in order to build PolyFit. It is thus important to understand how this parameter affects the query response time. Here, we use the form PolyFit-*deg* to represent the degree *deg* of PolyFit. Figure 11 shows the trends for the query response time for both COUNT (one key and two keys) and MAX (one key) queries, using the absolute error threshold $\varepsilon_{abs} = 100$. When we choose a larger degree *deg*, the polynomial function can provide better approximation for $F(k)$, and thus reduce the index size, which can reduce the response time for each query. However, the larger the degree *deg*, the larger the computation time for each node in PolyFit. Therefore, we can find that the response time increases (e.g., $deg = 3$ and 4 in Figure 11a), once we utilize a high degree *deg*. By default, in subsequent experiments, we choose deg = 2 for the COUNT query with a single key, and deg = 3 for the COUNT query with two keys and for the MAX query.

7.2.2 **Effect of *deg* on the construction time**. We further examine the construction time for PolyFit, varying the highest degree *deg* from 1 to 4 in the polynomial function (cf. Figure 12). Since a polynomial function with a higher degree can produce error guarantee for a longer interval I, i.e., $E(I) \leq \delta$, the GS method needs to call the LP solver with longer intervals (cf. line 4 in Algorithm 1), which can increase the construction time when using polynomial functions with higher degree *deg*.

7.3 Comparing with Error-Bounded Methods

In this section, we test the response time of the different methods that can fulfill the absolute and relative error guarantees. Here, we adopt the default settings for these methods (cf. Section 7.1) and use the datasets HKI, TWEET, and OSM for testing the performance of COUNT (single key), MAX (single key), and COUNT (two keys) queries, respectively. For Problem 1 (Q_{abs}), we fix the absolute error $\varepsilon_{abs} = 100$ and $\varepsilon_{abs} = 200$ for the experiments with one key and two keys, respectively. For Problem 2 (Q_{rel}), we fix the relative error $\varepsilon_{rel} = 0.01$. Table 5 shows the response time of different methods. Observe that PolyFit achieves the best performance for all the types of queries. For the COUNT query with two keys, PolyFit can achieve a speedup of at least two orders of magnitude over the existing methods.

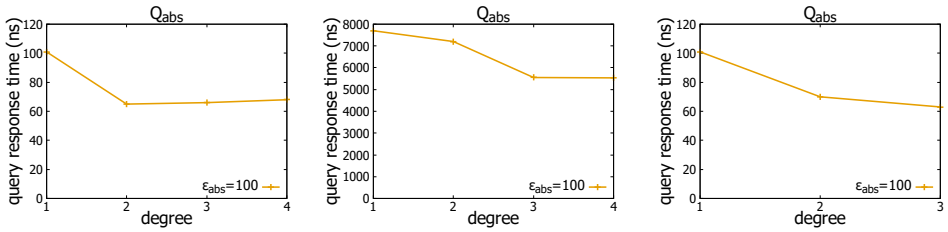
Table 5: Response time (nanoseconds) for all methods with error guarantees

Error guarantee	Q_{abs}			Q_{rel}		
Query type	COUNT	MAX	COUNT	COUNT	MAX	COUNT
# of keys	1	1	2	1	1	2
aR-tree	590	3592	357457	590	3592	357457
MRTree	565	182	385391	335	138	98919
RMI	568	n/a	n/a	579	n/a	n/a
FITing-tree	135	n/a	n/a	147	n/a	n/a
PGM	104	n/a	n/a	118	n/a	n/a
Polyfit	**68**	**63**	**5274**	**79**	**65**	**5299**

Sensitivity of ε_{abs} for COUNT query. We investigate how the absolute error ε_{abs} affects the response times of different methods. For the COUNT query with single key, we choose five absolute error values for testing, which are 100, 200, 400, 1000, and 2000. Observe from Figure 13a that since PolyFit, FITing-tree, and PGM can provide more compact index structures for the datasets, these methods can significantly improve the efficiency, compared with the traditional index structures, i.e., the aR-tree and the MRTree. In addition, due to the better approximation with nonlinear polynomial functions ($deg = 2$), PolyFit can achieve 1.33x to 6x speedups, over the existing learned-index structures, including RMI, FITing-tree, and PGM. For the COUNT query with two keys, we choose 200, 400, 800, 2000, and 4000 as the absolute error values for testing. Since the state-of-the-art learned index structures (RMI, FITing-tree, and PGM) can only support queries with a single key, we omit these methods in this experiment. Figure 13b shows that PolyFit achieves at least one order of magnitude speedups compared with the existing methods (aR-tree and MRTree), which is due to its compact index structure and query processing method.

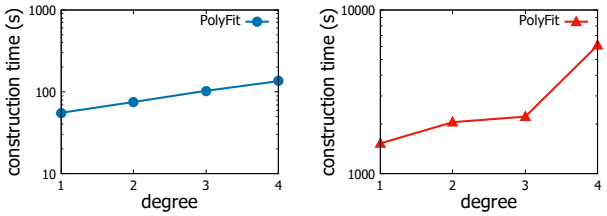
Sensitivity of ε_{rel} for COUNT query. We proceed to test how the relative error ε_{rel} affects the response time of the different methods. In this experiment, we choose five relative error values, which are 0.005, 0.01, 0.05, 0.1, and 0.2. Based on the more compact index structure, PolyFit is able to achieve better performance, compared with the existing methods (cf. Figure 14a). For the COUNT query with two keys, PolyFit significantly outperforms the existing methods, i.e., the aR-tree and the MRTree, by at least one order of magnitude (cf. Figure 14b).

Sensitivity of ε_{abs} and ε_{rel} for MAX query. In this experiment, we proceed to investigate how the absolute error ε_{abs} and



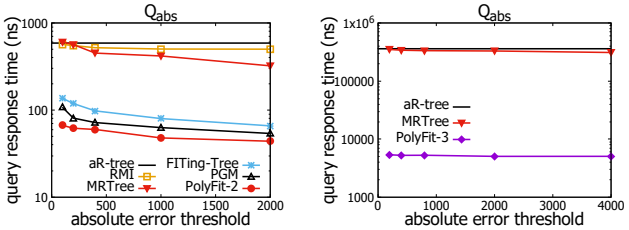
(a) COUNT query (single key) (b) COUNT query (two keys) (c) MAX query (single key)

Figure 11: Running time for COUNT (single key), COUNT (two keys), and MAX queries on TWEET, OSM, and HKI datasets, respectively, varying the degree *deg* of PolyFit



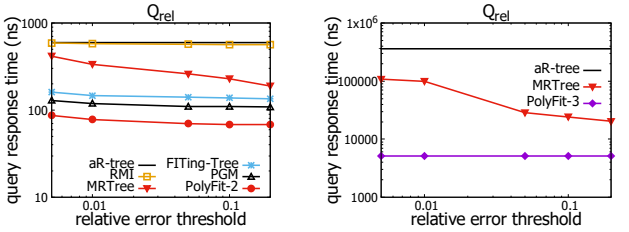
(a) COUNT query (single key) (b) COUNT query (two keys)

Figure 12: Index construction time of PolyFit for COUNT query with single key (using TWEET dataset) and two keys (using OSM dataset), varying the degree *deg*



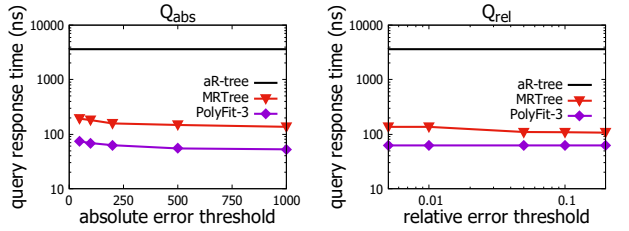
(a) COUNT query (single key) (b) COUNT query (two keys)

Figure 13: Response time for COUNT query in TWEET dataset (for single key) and OSM dataset (for two keys), varying the absolute error ε_{abs}



(a) COUNT query (single key) (b) COUNT query (two keys)

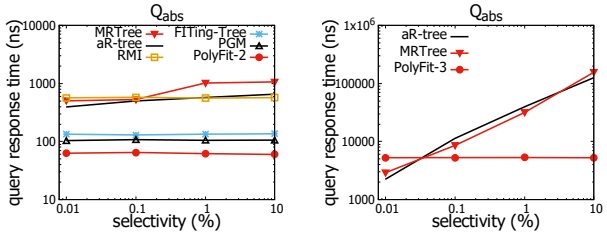
Figure 14: Response time for COUNT query in TWEET dataset (for single key) and OSM dataset (for two keys), varying the relative error ε_{rel}



(a) MAX query, varying ε_{abs} (b) MAX query, varying ε_{rel}

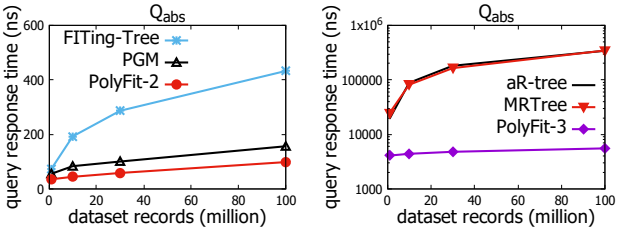
Figure 15: Response time for MAX query in HKI dataset

relative error ε_{rel} affect the efficiency performance of different



(a) COUNT query (single key) (b) COUNT query (two keys)

Figure 16: Response time for COUNT query in TWEET dataset (for single key) and OSM dataset (for two keys), varying the selectivity of the query



(a) COUNT query (single key) (b) COUNT query (two keys)

Figure 17: Response time for COUNT query in OSM dataset, varying the dataset size

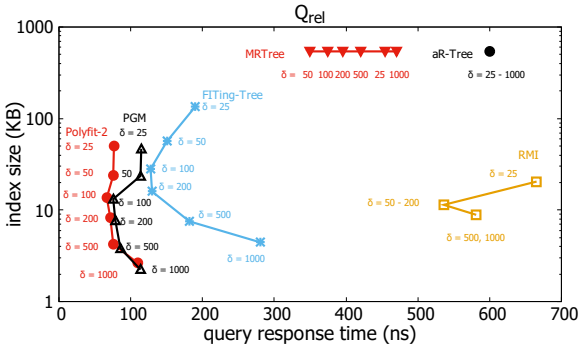


Figure 18: Trade-off between the query response time and index size of COUNT query (single key) in TWEET dataset, for Q_{rel} with $\varepsilon_{rel} = 0.01$, varying δ from 25 to 1000

methods. Observe from Figure 15, PolyFit can achieve at least 2x speedup, compared with other methods, even though the selected error is small.

Sensitivity of the selectivity for COUNT query. We further test the response time of the different methods, varying the selectivity of the COUNT query. Figure 16 shows that when we increase the selectivity of the COUNT query (i.e., each query covers a larger region), the query response time normally increases in different methods. Since all methods for the COUNT query with a single key have logarithmic time complexity, they are not sensitive to the selectivity (cf. Figure 16a). Unlike the single key case, both

the existing methods aR-tree and MRTree are sensitive to the selectivity, compared with PolyFit (cf. Figure 16b).

In both cases, PolyFit achieves better performance across different selectivities. Since the methods MRTree, aR-tree, and RMI always provide inferior efficiency in the single key case (cf. Figures 13a, 14a and 16a), compared with FITing-Tree, PGM and PolyFit, we omit their results in subsequent experiments.

Scalability to the dataset size. We proceed to test how the dataset size affects the efficiency of PolyFit and other methods. In this experiment, we choose the largest dataset OSM (with 100M records) for testing. Here, we focus on solving Problem 1 (Q_{abs}) for COUNT query, in which we adopt the default absolute errors, i.e., $\varepsilon_{abs} = 100$ and $\varepsilon_{abs} = 200$ for the cases in single key and two keys, respectively, and choose the latitude attribute as the key. To conduct this experiment, we choose five dataset sizes, which are 1M, 10M, 30M, and 100M. Figure 17 shows that PolyFit scales well with the dataset size and outperforms other methods.

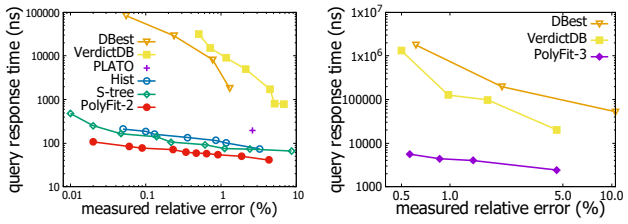
Trade-off between the query response time and index size. We proceed to investigate the trade-off between the query response time and index size of the different indexing methods. To conduct this experiment, we focus on Problem 2 (Q_{rel}) and choose 25, 50, 100, 200, 500, and 1000 as values of δ for testing. In Figure 18, since the changes to δ cannot affect the index construction methods of the aR-tree and MRTree, parameter δ cannot affect the index sizes of these two methods. We also notice that these index structures consistently provide inferior performance in terms of index size and query response time, compared with the FITing-tree, PGM, and the PolyFit methods. For the other methods, we can observe that the smaller the δ, the larger the index size and query response time. The reason is that smaller δ values lead to more leaf nodes in the index structures in the different methods (e.g., more intervals are generated by the GS method (cf. Algorithm 1) in PolyFit). On the other hand, if δ is too large, it is easier for an online query to violate the error condition for Q_{rel} (i.e., Lemma 5.2), and thus the query response time can also be larger. As such, all curves (except for the MRTree and aR-tree methods) in Figure 18 resemble the "C"-shape. In general, PolyFit-2 offers a better trade-off compared with other methods.

7.4 Comparing with Heuristic Methods

We compare the response time of PolyFit with other heuristic methods, which cannot fulfill deterministic error guarantees, i.e., Q_{abs} (cf. Problem 1) and Q_{rel} (cf. Problem 2). In this experiment, we adopt the default setting for the method PLATO [50], vary the bin size for the method Hist and vary the sampling size for the sampling-based methods, including S-tree, S2, VerdictDB, and DBest. Since S2 cannot achieve less than 100000ns query response time with 10% measured relative error, we omit the result of S2 in Figure 19a. In addition, we only report the results of the heuristic methods DBest and VerdictDB in Figure 19b, as the other heuristic methods cannot support COUNT queries with two keys (cf. Table 4). In these two figures, PolyFit yields the smallest query response time with similar relative error.

7.5 Comparing the Construction Time of All Methods

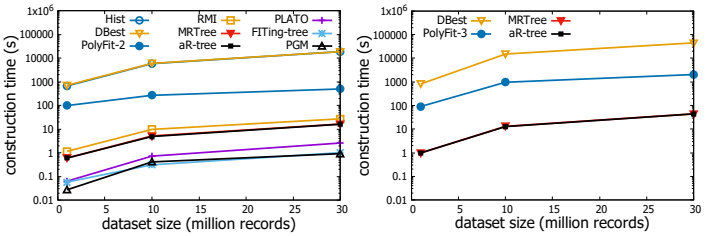
We proceed to investigate further how the construction times of all methods change across different dataset sizes. Here, we adopt the default degrees, i.e., $deg = 2$ and $deg = 3$, for the polynomial functions in the COUNT query with a single key and two keys, respectively. In Figure 20, PolyFit consistently achieves faster construction time than Hist and DBest. Although PolyFit may not achieve the fastest construction time, compared with



(a) COUNT query (single key) (b) COUNT query (two keys)

Figure 19: Response time between PolyFit and the heuristic methods for COUNT query with single key and two keys in TWEET and OSM datasets, respectively

some methods (e.g., the aR-tree and the MRTree), PolyFit takes less than 150s and 2500s (with default *deg*) in the construction stage with 1 million (TWEET) and 30 million records (OSM), respectively, which are acceptable in practice where the datasets are static during data analytics tasks.



(a) COUNT query (single key) (b) COUNT query (two keys)

Figure 20: Index construction time of methods for COUNT query with single key and two keys (using OSM dataset for both settings), varying the dataset size

8 CONCLUSION

In this paper, we study the range aggregate queries with two types of approximate guarantees, which are (1) absolute error guarantees (cf. Problem 1 (Q_{abs})) and (2) relative error guarantees (cf. Problem 2 (Q_{rel})). Unlike the existing methods, our work can efficiently support the most commonly used range aggregate queries (SUM, COUNT, MIN, MAX), fulfill the error guarantees, and support the setting of two keys.

In order to improve the efficiency of computing these queries, we utilize several polynomial functions to fit the data points and then build the compact index structure PolyFit on top of these polynomial functions. An experimental study shows that PolyFit can achieve significant speedups compared with existing learned-index methods and other traditional exact/ approximate methods for different query types. In particular, we can achieve at most 5μs query response time in a dataset with 30 million records, which cannot be achieved by the state-of-the-art methods.

In the future, we plan to further develop advanced techniques to improve the efficiency of constructing PolyFit, in order to handle updates of records in large-scale datasets. In addition, we aim to extend our methods to support other fundamental analytics operations, including standard deviation, median, etc. Moreover, we plan to investigate how to utilize the idea of PolyFit to further improve the efficiency of other types of statistics and machine learning models, e.g., kernel density estimation [16, 18], and support vector machines [17, 18].

REFERENCES

[1] CPLEX performance tuning for linear programs. https://www.ibm.com/support/pages/node/397127#Item4.

[2] Diagnosing ill conditioning. https://www.ibm.com/support/pages/node/397063.
[3] Foursquare API. https://developer.foursquare.com/.
[4] Foursquare statistics. https://99firms.com/blog/foursquare-statistics/#gref/.
[5] Hong Kong 40 Index 2018. https://www.dukascopy.com/swiss/english/marketwatch/historical/. [Online; accessed 20-Dec-2019].
[6] STX B+ Tree. https://panthema.net/2007/stx-btree/. [Online; accessed 11-Jan-2019].
[7] OpenStreetMap dataset. https://registry.opendata.aws/osm/, 2019. [Online; accessed 19-May-2019].
[8] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD*, pages 181–192, 1999.
[9] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.
[10] C. Anagnostopoulos and P. Triantafillou. Learning to accurately count with query-driven predictive analytics. In *BigData*, pages 14–23, 2015.
[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.
[12] D. Bartholomew. *MariaDB cookbook*. Packt Publishing Ltd, 2014.
[13] S. M. Beitzel, E. C. Jensen, A. Chowdhury, D. Grossman, and O. Frieder. Hourly analysis of a very large topically categorized web query log. In *SIGIR*, pages 321–328, 2004.
[14] J. L. Bentley and A. C. Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976.
[15] K. Chan and A. W. Fu. Efficient time series matching by wavelets. In *ICDE*, pages 126–133, 1999.
[16] T. N. Chan, R. Cheng, and M. L. Yiu. QUAD: quadratic-bound-based kernel density visualization. In *SIGMOD*, pages 35–50. ACM, 2020.
[17] T. N. Chan, L. H. U, R. Cheng, M. L. Yiu, and S. Mittal. Efficient algorithms for kernel aggregation queries. *IEEE TKDE*, pages 1–1, 2020.
[18] T. N. Chan, M. L. Yiu, and L. H. U. KARL: fast kernel aggregation queries. In *ICDE*, pages 542–553, 2019.
[19] L. Chen, G. Cong, X. Cao, and K.-L. Tan. Temporal spatial-keyword top-k publish/subscribe. In *ICDE*, pages 255–266, 2015.
[20] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD*, pages 1115–1118, 2010.
[21] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
[22] M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008.
[23] K. Delaney. *Inside Microsoft SQL Server 2000*. Microsoft Press, 2000.
[24] A. Eldawy, M. F. Mokbel, S. Al-Harthi, A. Alzaidy, K. Tarek, and S. Ghani. SHAHED: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. In *ICDE*, pages 1585–1596, 2015.
[25] H. Elmeleegy, A. K. Elmagarmid, E. Cecchet, W. G. Aref, and W. Zwaenepoel. Online piece-wise linear approximation of numerical streams with precision guarantees. *VLDB*, 2(1):145–156, 2009.
[26] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.
[27] P. Ferragina and G. Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, 13(8):1162–1175, 2020.
[28] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. Fiting-tree: A data-aware index structure. In *SIGMOD*, pages 1189–1206, 2019.
[29] M. Garofalakis and P. B. Gibbons. Wavelet synopses with error guarantees. In *SIGMOD*, pages 476–487, 2002.
[30] M. N. Garofalakis and P. B. Gibbons. Probabilistic wavelet synopses. *ACM Trans. Database Syst.*, 29:43–90, 2004.
[31] J. L. Gearhart, K. L. Adair, R. J. Detry, J. D. Durfee, K. A. Jones, and N. Martin. Comparison of open-source linear programming solvers. *Sandia National Laboratories, SAND2013-8847*, 2013.
[32] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In *SIGMOD*, pages 463–474, 2000.
[33] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *VLDBJ*, 14(2):137–154, 2005.
[34] P. J. Haas, J. F. Naughton, and A. N. Swami. On the relative cost of sampling for join selectivity estimation. In *PODS*, pages 14–24, 1994.
[35] P. J. Haas and A. N. Swami. Sequential sampling procedures for query size estimation. In *SIGMOD*, pages 341–350, 1992.
[36] S. Han, H. Wang, J. Wan, and J. Li. An iterative scheme for leverage-based approximate aggregation. In *ICDE*, pages 494–505, 2019.
[37] M. Heimel, M. Kiefer, and V. Markl. Self-tuning, GPU-accelerated kernel density models for multidimensional selectivity estimation. In *SIGMOD*, pages 1477–1492, 2015.
[38] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *SIGMOD*, pages 73–88, 1997.
[39] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658, 2004.
[40] S. K. Jensen, T. B. Pedersen, and C. Thomsen. ModelarDB: modular model-based time series management with spark and cassandra. *PVLDB*, 11(11):1688–1701, 2018.
[41] E. J. Keogh. Fast similarity search in the presence of longitudinal scaling in time series databases. In *ICTAI*, pages 578–584, 1997.
[42] E. J. Keogh, S. Chu, D. M. Hart, and M. J. Pazzani. An online algorithm for segmenting time series. In *ICDM*, pages 289–296, 2001.
[43] E. J. Keogh and M. J. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In *KDD*, pages 239–243, 1998.
[44] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD*, pages 489–504, 2018.
[45] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, pages 401–412, 2001.
[46] Y. T. Lee and A. Sidford. Efficient inverse maintenance and faster algorithms for linear programming. In *FOCS*, pages 230–249, 2015.
[47] D. Leenaerts and W. van Bokhoven. *Piecewise Linear Modeling and Analysis*. Springer US, 2013.
[48] Z. Li, T. N. Chan, M. L. Yiu, and C. S. Jensen. Polyfit: Polynomial-based indexing approach for fast approximate range aggregate queries. *CoRR*, abs/2003.08031, 2020.
[49] L. Lim, M. Wang, and J. S. Vitter. Sash: A self-adaptive histogram set for dynamically changing workloads. In *VLDB*, pages 369–380, 2003.
[50] C. Lin, E. Boursier, and Y. Papakonstantinou. Plato: approximate analytics over compressed time series with tight deterministic error guarantees. *PVLDB*, 13(7):1105–1118, 2020.
[51] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *SIGMOD*, pages 1–11, 1990.
[52] C. A. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distribution of column values. In *VLDB*, pages 240–251, 1988.
[53] Q. Ma and P. Triantafillou. DBEst: Revisiting approximate query processing engines with machine learning models. In *SIGMOD*, pages 1553–1570, 2019.
[54] A. Marcus, M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, and R. C. Miller. Processing and visualizing the data in tweets. *SIGMOD Record*, 40(4):21–27, 2011.
[55] V. Markl, P. J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. M. Tran. Consistent selectivity estimation via maximum entropy. *VLDBJ*, 16(1):55–76, 2007.
[56] B. Momjian. *PostgreSQL: introduction and concepts*. Addison-Wesley New York, 2001.
[57] M. Muralikrishna and D. J. DeWitt. Equi-depth multidimensional histograms. In *SIGMOD*, pages 28–36, 1988.
[58] T. Palpanas, M. Vlachos, E. J. Keogh, and D. Gunopulos. Streaming time series summarization using user-defined amnesic functions. *IEEE TKDE*, 20(7):992–1006, 2008.
[59] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *SSTD*, pages 443–459, 2001.
[60] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. VerdictDB: Universalizing approximate query processing. In *SIGMOD*, pages 1461–1476, 2018.
[61] Y. Park, S. Zhong, and B. Mozafari. Quicksel: Quick selectivity learning with mixture models. In *SIGMOD*, pages 1017–1033, 2020.
[62] I. Popivanov and R. J. Miller. Similarity search over time-series data using wavelets. In *ICDE*, pages 212–221, 2002.
[63] D. Rafiei. On similarity-based queries for time series data. In *ICDE*, pages 410–417, 1999.
[64] C. Ré and D. Suciu. Understanding cardinality estimation using entropy maximization. In *PODS*, pages 53–64, 2010.
[65] M. Riondato, M. Akdere, U. Çetintemel, S. B. Zdonik, and E. Upfal. The vc-dimension of sql queries and selectivity estimation through sampling. In *ECML PKDD*, pages 661–676, 2011.
[66] F. Savva, C. Anagnostopoulos, and P. Triantafillou. Aggregate query prediction under dynamic workloads. In *BigData*, pages 671–676, 2019.
[67] I. N. Stewart. *Galois theory*. CRC Press, 2015.
[68] H. To, K. Chiang, and C. Shahabi. Entropy-based histograms for selectivity estimation. In *CIKM*, pages 1939–1948, 2013.
[69] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *SIGMOD*, pages 193–204, 1999.
[70] H. Wang, X. Fu, J. Xu, and H. Lu. Learned index for spatial queries. In *MDM*, pages 569–574, 2019.
[71] A. Wasay, X. Wei, N. Dayan, and S. Idreos. Data canopy: Accelerating exploratory statistical analysis. In *SIGMOD*, pages 557–572, 2017.
[72] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.
[73] X. Yu, Y. Xia, A. Pavlo, D. Sanchez, L. Rudolph, and S. Devadas. Sundial: harmonizing concurrency control and caching in a distributed oltp database management system. *PVLDB*, 11(10):1289–1302, 2018.
[74] X. Yun, G. Wu, G. Zhang, K. Li, and S. Wang. Fastraq: A fast approach to range-aggregate queries in big data environments. *IEEE Trans. Cloud Computing*, 3(2):206–218, 2015.

Shift-Table: A Low-latency Learned Index for Range Queries using Model Correction

Ali Hadian
Imperial College London

Thomas Heinis
Imperial College London

ABSTRACT

Indexing large-scale databases in main memory is still challenging today. Learned index structures — in which the core components of classical indexes are replaced with machine learning models — have recently been suggested to significantly improve performance for read-only range queries.

However, a recent benchmark study shows that learned indexes only achieve limited performance improvements for real-world data on modern hardware. More specifically, a learned model cannot learn the micro-level details and fluctuations of data distributions thus resulting in poor accuracy; or it can fit to the data distribution at the cost of training a big model whose parameters cannot fit into cache. As a consequence, querying a learned index on real-world data takes a substantial number of memory lookups, thereby degrading performance.

In this paper, we adopt a different approach for modeling a data distribution that complements the model fitting approach of learned indexes. We propose *Shift-Table*, an algorithmic layer that captures the micro-level data distribution and resolves the local biases of a learned model at the cost of at most one memory lookup. Our suggested model combines the low latency of lookup tables with learned indexes and enables low-latency processing of range queries. Using Shift-Table, we achieve a speedup of 1.5X to 2X on real-world datasets compared to trained and tuned learned indexes.

1 INTRODUCTION

Trends in new hardware play a significant role in the way we design high-performance systems. A recent technological trend is the divergence of CPU and memory latencies, which encourages decreasing random memory access at the cost of doing more compute on cache-resident data [25, 42, 44].

A particularly interesting family of methods exploiting the memory/CPU latency gap are learned index structures. A learned index uses machine learning instead of algorithmic data structures to learn the patterns in data distribution and exploits the trained model to carry out the operations supported by an algorithmic index, e.g., determining the location of records on physical storage [7, 12, 18, 24, 25, 29]. If the learned index manages to build a model that is compact enough to fit in processor cache, then the results can ideally be fetched with a single access to main memory, hence outperforming algorithmic structures such as B-trees and hash tables.

In particular, learned index models have shown a great potential for range queries, e.g., retrieving all records where the key is in a certain range $A <$ key $< B$. To enable efficient retrieval of range queries, range indexes keep the records physically sorted. Therefore, retrieving the range query is equivalent to finding the first result and then sequentially scanning the records to retrieve the entire result set. Therefore, processing a range query $A <$ key $< B$ is equivalent to finding the first result, i.e., the smallest key in the dataset that is greater than or equal to A (similar to `lower_bound(A)` in the C++ Library standard). A learned index can be built by fitting a regression model to the cumulative distribution function (CDF) of the key distribution. The learned CDF model can be used to determine the physical location where the lower-bound of the query resides, i.e., $\texttt{pos}(\text{A}) = N \times F_\theta(\text{A})$ where N is the number of keys and F_θ is the learned CDF model with model parameters θ.

Learned indexes are very efficient for sequence-like data (e.g., machine-generated IDs), as well as synthetic data sampled from statistical distributions. However, a recent study using the Search-On-Sorted-Data benchmark (SOSD) [22] shows that for real-world data distributions, a learned index has the same or even poorer performance compared to algorithmic indexes. For many real-world data distributions, the CDF is too complex to be learned efficiently by a small cache-resident model. The data distribution of real-world data has "too much information" to be accurately represented by a small machine-learning model, while an accurate model is needed for an accurate prediction. One can of course use smaller models that fit in memory with the cost of lower prediction accuracy, but will end up in searching a larger set of records to find the actual result which consequently increases memory lookups and degrades performance. Alternatively, a high accuracy can be achieved by training a bigger model, but accessing the model parameters incurs multiple cache misses and also increases memory lookups, reducing the margins for performance improvement.

In this paper, we address the challenge of using learned models on real-world data and illustrate how the micro-level details (e.g., local variance) of a cumulative distribution can dramatically affect the performance of a range index. We also argue that a pure machine learning approach cannot shoulder the burden of learning the fine-grained details of an empirical data distribution and demonstrate that not much improvement can be achieved by tuning the complexity or size thresholds of the models.

We suggest that by going beyond mere machine learning models, the performance of a learned index architecture can be significantly improved using a complementary enhancement layer rather than over-emphasizing on the machine learning tasks. Our suggested layer, called *Shift-Table* is an algorithmic solution that improves the precision of a learned model and effectively accelerates the search performance. Shift-Table, targets the micro-level bias of the model and significantly improves the accuracy, at the cost of only one memory lookup. The suggested layer is optional and applied after the prediction; it can hence be switched on or off without re-training the model.

Our contributions can be summarized as follows:

- We identify the problem of learning a range index for real-world data, and illustrate the difficulty of learning from this data.

- We suggest the Shift-Table approach for correcting a learned index model, which complements a valid (monotonically increasing) CDF model by correcting its error.
- We show how, and in which circumstances, the suggested methods can be used for best performance.
- We suggest cost models that determine whether the Shift-Table layer can boost performance.
- The experimental results show that our suggested method can improve existing learned index structures and bring stable and almost-constant lookup time for real-world data distributions. Our enhancement layer achieves up to 3X performance improvement over existing learned indexes. More interestingly, we show that for non-skewed distributions, the Shift-Table layer is effective enough to help a dummy linear model outperform the state of the art learned indexes on real-world datasets

2 MOTIVATION

2.1 Lookup Cost for Learned Models

In modern hardware, the lookup times of in-memory range indexes and the binary search algorithm are mainly affected by their memory access pattern, most notably by how the algorithm uses the cache and the Last-Level-Cache (LLC) miss rate.

Processing a range query in a learned index has two stages: 1) **Prediction**: Running the learned model to predict the location of the first result for the range query; and 2) **Local search** (also known as *last-mile search*): searching around the predicted location to find the actual location of the first result. Figure 1a shows common search methods for the local search. If the learned model can determine a guaranteed range area around the predicted position, one can perform binary search. Otherwise, exponential or linear search should be used, starting from the predicted position.

A cache miss in a learned index can occur in the first stage for accessing the parameters of the model (if the model is too big to fit in cache), or in stage two for the local search. Key in understanding the cost of a learned index is that local search is done entirely over non-cached blocks of memory. A learned index built over millions of records could predict the location of records with an error of, say, 1000 records and yet achieve no performance gain over binary search algorithms or algorithmic indexes. This is because while the learned index fits the models in cache, its algorithmic competitors also fit the frequently-accessed parts of the data in cache, which limits the potential for improvement for a learned index.

2.2 Lookup Cost for Algorithmic Indexes

Classical algorithms, such as binary search, can be seen as a hierarchy of [non-learned] models, which take the middle-point as its parameter and predicts (accurately) which direction the search should follow. Specifically for the first few steps of binary search where the middle-points usually reside in cache, the functionality of binary search is the same as a learned model from a performance point of view.

In a pure binary search on the entire data, the first set of memory locations accessed by the algorithm (i.e., the median, quarters, etc.) will already be in the CPU cache after a few lookups. Therefore, the major bottleneck in binary search is for the latter stages of search where the middle elements are not in cache, causing last-level-cache (LLC) misses. Figure 1b shows a schematic illustration of how caching accelerates binary search.

In basic implementations of binary search, the "hot keys" are cached with their payload and nearby records in the same cache line, which wastes cache space. Binary search thus uses the cache poorly and there are more efficient algorithmic approaches whose performance is not sensitive to data distributions.

Cache-optimized versions of binary search, e.g., a binary search tree such as FAST [21], a read-only search tree that co-locates the hot keys but still follows the simple bisecting method of binary search, are up to 3X faster than binary search [22]. This is because FAST keeps more hot keys in the cache and hence it needs to scan a shorter range of records in the local search phase (cache-non-resident iterations of the search).

2.3 Preliminary Experimental Analysis

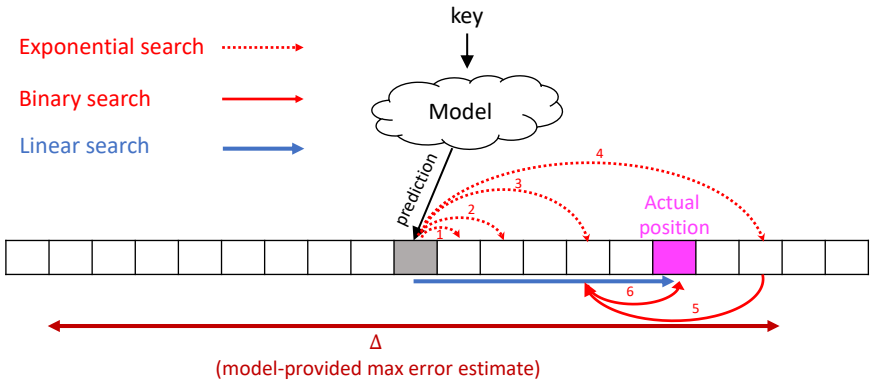
For a tangible discussion and to elaborate on the real cost of a learned model, we provide a micro-benchmark that measures the cost of errors in a learned index. We use the experimental configuration used in the SOSD benchmark [22], i.e., searching over 200M records with 32-bit keys and 64-bit payloads. Figure 2a shows the lookup time of the second phase (local search) in a learned model for different prediction errors. We include the lookup times for binary search, as well as FAST [21], over the whole array of 200M keys.

We are interested to see that if the position predicted by a learned index, say $\texttt{predicted\_pos}(x)$, has an error Δ, then how long does it take in the local phase to find the correct record. Thus, for each query x_i, we pre-compute the 'output' of the learned index with error Δ, i.e., $[\texttt{predicted\_pos}(x_i) \pm \Delta]$, and then run the benchmark given $\{x_i, [\texttt{predicted\_pos}(x_i) \pm \Delta]\}$ tuples.

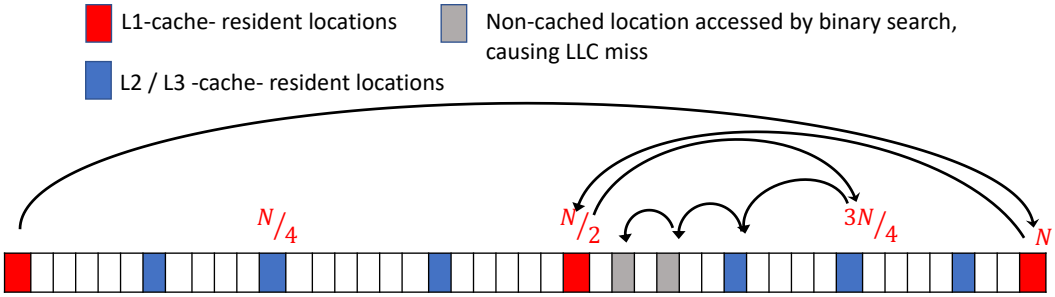
As shown in Figure 2a, if the error of the model is more than ~300 records on average, then FAST outperforms the learned model (with linear or exponential local search). Even if the learned model can give a guaranteed range around the predicted point to guide the local search and enable binary search, FAST outperforms it if the error exceeds 1000 records. The same trend can be seen for the LLC miss rates in Figure 2b.

Note that this micro-benchmark over-estimates the maximum error that the learned index can have because we only compare the time of *local search phase* in a learned index with the *total search time* of FAST and binary search. Considering the time taken to execute the model for predicting the location, a learned model needs to have a much lower error to compete with the generic, reliable, and distribution-independent algorithms such as binary search and FAST. For example, FAST takes 200 nanoseconds to search a key in the entire 200M-key dataset. If a learned index takes, say, 120 nanoseconds to run (for accessing model parameters and computing the prediction), then the local search can take at most 80 nanoseconds so that the learned index can outperform FAST, which means that the prediction error (Δ) must be less than 16 records (based on Figure 2a).

Tuning the learned index for a balance of model size and accuracy is a challenging task. Improving the local search time requires using a more accurate model with a higher learning capacity and more parameters. However, accessing such a big model typically incurs further cache misses during model execution, and consequently the lookup time. Therefore, if the data distribution cannot be learned efficiently with a small memory footprint (fitting into cache), outperforming cache-efficient algorithmic indexes is very challenging. This is indeed the case for most real-world datasets that cannot be modelled accurately with a small-sized model.



(a) Different "last-mile" search methods (performed after location prediction) in learned index. The locations predicted by the model depend on the query and are not known in advance. Since the last-mile search algorithms need to access different memory locations for each query, they cannot exploit the processor cache and the search algorithm incurs multiple cache misses



(b) Schematic illustration of processor caching in binary search. The locations accessed by the very early stage of binary search, such as the min, max, and the midpoint, are frequently accessed and available in the L1 cache. Further steps of binary access locations that are less frequently accessed and fit on lower levels of the memory hierarchy. Therefore, a deterministic search algorithm like binary search enjoys a high cache hit rate

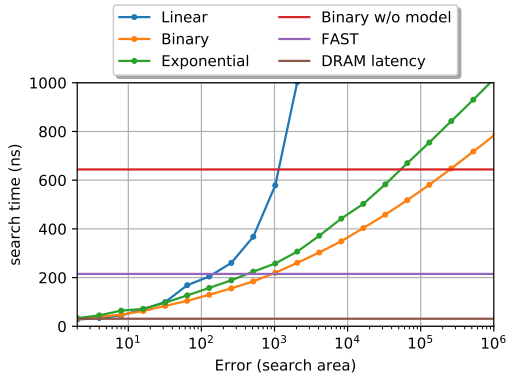
Figure 1: Comparison of patterns in binary search (partially cached) and local search in learned indices (non-cached).

2.4 Difficulty of Learning Real-world Data

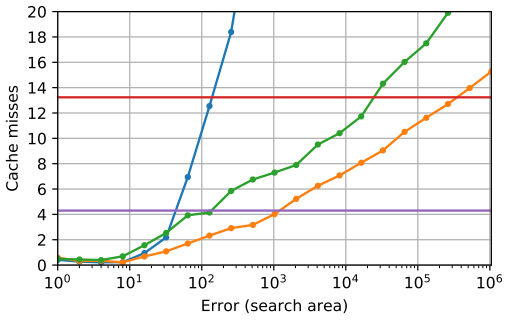
To use a learned index in a production system, it is essential to identify when learned indexes fail to achieve superior performance and what aspects of the data distribution contributes to the performance of a learned index model. We realized that a major challenge in understanding learned indexes is that the common practices of performance evaluation for indexing algorithms are misleading for learned indexes. For example, it is common to use the uniform and skewed distributions (such as log-normal) as arguably the two best- and worst-case extremes for a search task [25]. However, for evaluating search over sorted read-only data, the difficulty of the task is determined by the *unpredictability* of the data, which is not necessarily a factor of skewness or shape parameter of the data distribution. As we will show in this section, most statistical distributions are much easier to model than real-world data.

Distributions that matter. An interesting observation from the SOSD benchmark results is that even for datasets that have the same background distribution, e.g., both closely match a uniform distribution, the performance of a learned model can vary significantly, depending on the fine-grained details in the empirical CDFs. For example, consider Figures 3a and 3b, which represent two CDFs that are both close to uniform. The uniform data (`uden64` [22]) is comprised of dense integers that are synthetically sampled from a uniform distribution, and Facebook (`face64` [22]) is a Facebook user ID dataset. While both datasets match closely with the uniform distribution, face64 is significantly harder to model due to its fine-grained details in the CDF. The lookup time of learned indexes (both RMI and Radix-Splines) for face64 is 6-7× higher than that of uden64 (see Table 2) because there are many micro-level details (unpredictability) in the CDF, hence a huge model with a high learning capacity is needed to fit the CDF accurately. Using the RMI learned index, for example, the uden64 data is easily modelled with a simple line (two parameters) with near-zero error, while the best architecture found by the SOSD benchmark for modelling the face64 data is a hierarchy of two linear models, a huge model (136MB), with an average error of 202 records.

Generally speaking, real-world datasets are more difficult to learn compared to synthetic ones and the learned index built over them is not significantly faster than the algorithmic rivals. The main question remains what distinguishes a real-world data from a synthetic one? Consider the four distributions in Figure 3, where Figures 3a, 3c are synthetic (generated from uniform and log-normal distributions), and Figures 3b, 3d are real-world data. The mini-chart inside each CDF highlights the distribution in a small sub-range, i.e., a "zoomed-in" view of the CDF. For the synthetic data, the CDF is very smooth in any short sub-range of the whole CDF. Synthetic data (such as uniform, normal and log-normal) are built using a cumulative density function that is derivable, meaning that the at any small sub-range, the shape of the CDF is close to a straight line with a slope that is close to the derivation of the underlying CDF in that range. Such a



(a) Lookup time



(b) Cache misses

Figure 2: Cost of local search in a learned index

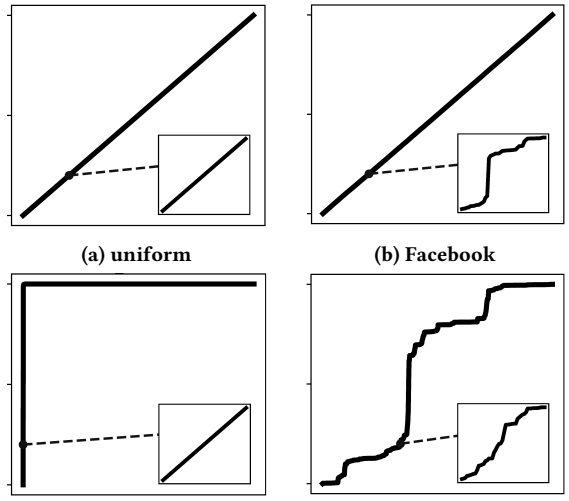
smooth CDF has less information to be compressed into a model. For example, a learned index model based on linear splines can accurately fit the whole CDF by fitting each part of the CDF to a line. Even for very skewed distributions, such as log-normal, the data is so predictable that it can be easily fitted to simple, linear models.

Real-world data, however, is much less predictable and has a much higher level of complexity in its patterns. Even if an ideal learning algorithm is used to model the real-world data, the model itself needs to be very big because the compressed version of the CDF (to be stored as a model) is still very big.

This explains why state-of-the-art learned indexes perform extremely well for datasets that are synthetically generated from a statistical distribution (such as uniform, normal, and log-normal), but perform comparably poor for real-world data that even almost match (shapewise) with those synthetic distributions [22]. On real-world datasets, learned indexes have a high cache miss rate and lookup time, contrary to their primary goal of having fewer cache misses.

Using learned models is beneficial when they are 1) accurate enough to predict a position within the same cache line that contains the data point, otherwise the lookup time will be adversely affected due to multiple cache misses, and 2) compact enough to fit in cache and not to cause LLC misses. With this in mind, we can argue that a pure machine-learning approach might fail to "learn the data perfectly" and "fit the model in cache" simultaneously, specifically in case of real-world datasets that contain a lot of underlying patterns like spikes and generally noise.

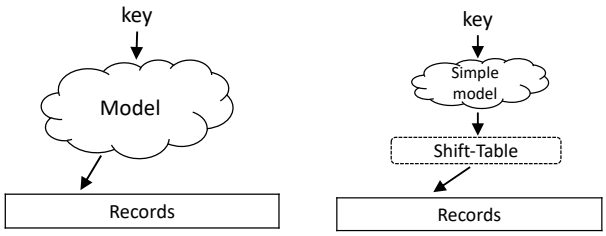
As a consequence, learned models are crucial to indexing but they cannot shoulder the burden of indexing the data alone. We hence suggest an algorithmic layer that can mitigate the difficulty



(a) uniform (b) Facebook

(c) Lognormal (d) OSMC

Figure 3: Example distributions with different complexities in micro and macro levels



(a) Learned index (b) Model + Shift-Table

Figure 4: Leveraging correction layers to a learned index

of learning the data distribution. In this approach, the learned model is allowed to learn an semi-accurate, small model that learns the holistic shape of the distribution, and the fine-tuned modelling is provided by the algorithmic layers.

2.5 Model Correction

While learned index models are powerful tools for describing a data distribution in a compact representation, merely focusing on learning a highly-accurate model does not necessarily lead to a high-performance index. In this paper, we suggest a new approach for boosting existing learned models with additional layers, specifically developed with hardware costs in mind.

The suggested helping layers add a small overhead when executing queries, but significantly reduce the overall lookup time of the learned index. The suggested layers are very powerful and consequently allow for using more lightweight models, yet ideally avoid computationally-expensive algorithms for training.

As Figure 4 illustrates, in addition to the learned index model we add a correction layer, an optional component, that can be added to improve the performance. We explore the potential of correction layers in the next sections.

3 SHIFT-TABLE

A learned model predicts a relative position $F_\theta(x)$ for a given query x. To calculate the position of the result, the estimated

relative position is multiplied by the number of keys, and truncated to an integer (the index), hence the predicted position is $[NF_\theta(x)]$. The actual position of the record, however, is $NF(x)$ where $F(x)$ is the empirical CDF of the data points, and N is the data size. Therefore, the result is $NF(x) - [NF_\theta(x)]$ records ahead of the predicted position. We identify $NF(x) - [NF_\theta(x)]$ as the *drift* of F_θ at key x, which is the signed error of the prediction, as opposed to the absolute error.

The idea of the Shift-Table layer is to have a lookup table that contains the drift values so that the drift of the prediction can be corrected. Capturing the drift for every value of x requires an auxiliary index, which is not feasible. However, we can use the *output* of the learned index model ($[NF_\theta(x)]$), which is in the range of $[0, N]$, and we construct a mapping from each possible output of the model, say k, to "how far ahead is the actual record if model predicts k's record", so that we can correct the predictions using this mapping. This means that for each prediction, we only need an extra lookup of k in a fixed array of size N.

To build the Shift-Table layer, we first partition the keys $x_0, \cdots, x_{N-1}$ into N partitions. We define P_k as the set of keys for which the model predicts k as the position:

$$P_k = \{x \mid [NF_\theta(x)] = k\} \quad (1)$$

Each of the indexed keys in P_k has an index, say $NF(x)$ and a prediction $k = [NF_\theta(x)]$. For each partition, we extract two parameters that specify the range for local search, namely Δ_k and C_k. Δ_k is defined as:

$$\Delta_k = \min\left(NF(x) - k\right) \quad \forall x \in P_k \quad (2)$$

which indicates that if the predicted location is k, the search should be started at point $k+\Delta_k$. Also, $C_k = |P_k|$ is the cardinality of P_k, i.e., the number of indexed keys for which the prediction predicts the k'th record, in other words, the length of the area that has to be searched in the local search phase.

To correct the prediction, we first compute the predicted position $k = [NF_\theta(x)]$, and then perform local search in the range of $[k + \Delta_k, k + \Delta_k + C_K - 1]$.

The number of partitions depends on the range of the output of the learned index, which should be $0, N)$. Therefore, The $<\Delta_k, C_k>$ pairs: pairs are stored in a single array of size N, so that the correction can be done using a single lookup into the array of pairs.

A Shift-Table layer is depicted in Figure 5. The index contains 100 elements in range [0,999]. The CDF model is a simple model: $F_\theta(x) = x/1000$, hence the prediction is simply $k = [x/10]$. If the query is 771, for example, the prediction of the model is $k = 77$. The correction information are $\Delta_{77} = -41$ and $C_{77} = 2$, which indicates that the result is -41 records ahead of the prediction, and the search area is of length 2. Therefore, the local search is performed on the indexes of range $[36, 37]$.

Algorithm 1 shows how Shift-Table is used to accelerate query processing. The Shift-Table layer reduces the prediction error of the model, but incurs an additional memory lookup.

Algorithm 1 Search with direct-mapped learned index

```
1:  procedure FIND_LOWER(q, model, Shift-Table)
2:      pos = model.predict(q)
3:      pos = Shift_Table.mapping[pos].startPoint
4:      range = Shift_Table.mapping[pos].range
5:      if range < linear_to_binary_threshold then
6:          pos = LinearSearch(start=data[pos],range)
7:      else
8:          pos = BinarySearch(start=data[pos],range)
9:      end if
10:     return pos
11: end procedure
```

3.1 Querying non-indexed keys

If the query is on the indexed keys, the result is in range $[k + \Delta_k\ ,\ k + \Delta_k + C_K - 1]$. In Figure 5, for example, querying 771 and 782 points to the correct range that contains the result. However, if the query is not among the indexed keys, then the query is either within the range, or in the position just after the range (at data$[k+\Delta_k+C_K]$. For example, in Figure 5, the record corresponding to queries 778 and 781 is the same, though the aforementioned model ($k = [q/10]$), maps 778 to range $[36, 37]$, and 781 to $[38, 39]$. In both cases, however, the local search algorithm (either binary or linear search) within the range computes the correct position of the result (i.e., 38). Notably for $q = 778$, a typical local search implementation realizes that the query is greater than the largest value in range and returns the first index right after the range of $[36, 37]$, which is 38.

Another issue that can arise for non-indexed keys is when the predicted position P_k has an empty partition that none of the indexed keys belongs to. In Figure 5, if the query is 15, then the predicted position is $k = [15/10] = 1$, but P_1 is empty because the model does not predict position 1 for any of the indexed keys. If the query is predicted to be in an empty partition, the result is the first record in the next non-empty partition, e.g., the result of query=15 is record 3. To make the Shift-Table layer consistent for the empty partitions, we put pseudo values for Δ, C in the mapping layer such that they refer to the same range as the next existing partition. If $P_{k^\varnothing}$ is an empty partition and P_k is the first non-empty partition after $P_{k^\varnothing}$, then $C_{k^\varnothing} = C_k$ and $\Delta_{k^\varnothing} = \Delta_k + (k - k^\varnothing)$. The pseudo Δ, C-values are depicted using dashed arrows in Figure 5.

3.2 CDF and duplicate values

It should be noted that the *empirical CDF* function, i.e., $F(X) = P(X \leq x)$ does not exactly identify the result of a range query on x. In this paper, we use the CDF (F(x)) notation as the index of the result corresponding to x. We consider range queries of type (`key <= query`), hence the CDF for a point x is the relative position of the *first* key in the indexed keys, as the range is scanned towards the right. More precisely, we assume that $NF(x_0) = 0$ and $NF(x_{N-1}) = N - 1$ (for the last key).

A range learned index built for a specific comparison operator, say $x \leq q$, can be used for other operators ($\geq, >,$, etc.) with a brief left/right scan. However, if there are too many duplicates in the indexed data, then the the performance of the learned index will be worse for queries that do not match the presumed definition of F(X). In such cases, it is more efficient to use the specific definition of $F(x)$ that reflects the position of the result of the query in the most common type of constraint in the queries. For example, if most of the queries are of type $x >= q$, then $F(x)$ should be defined such that $NF(x)$ identifies the index of the last key among the duplicate values.

3.3 Building the Shift-Table layer

Algorithm 2 describes how the mapping of the Shift-Table layer is built. In the first stage, it computes the Δ, C values and updates for the non-empty partitions, i.e., P_ks for which at least one of

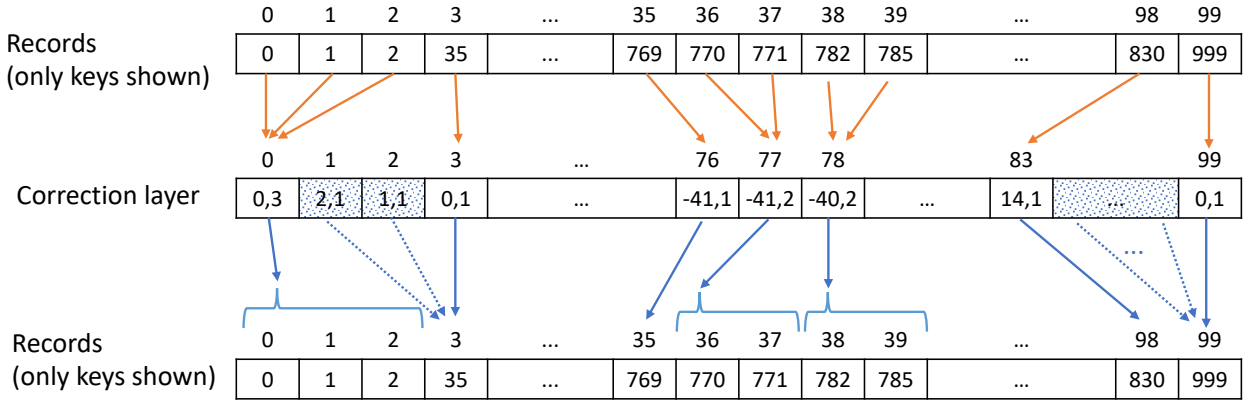


Figure 5: Shift-Table

the indexed keys is mapped to k. In the second stage, a backward traversal is performed on the Shift-Table layer and the compute the pseudo-values for the empty partitions (Algorithm 2, lines 10–14). Starting from the last entry, a pseudo-partition has the same count (C) as the first non-empty partition on its right side, but the shift Δ is adjusted so that they both point the the same region for local search.

The computational complexity of building the Shift-Table layer is $O(N) \times O(F_\theta)$ to compute the drifts and updating the mapping, as it only traverses the data and the Shift-Table layer once. In case that running the model is expensive, model executions can be parallelized for faster execution.

Algorithm 2 Building the Shift-Table layer

```
1:  procedure SHIFT-TABLE_BUILD(model (F_θ), data)
2:      Shift-Table = Array of tuples <Δ, C>, all set to zero
3:      for all x ∈ data do
4:          pos = NF(x)                          ▷ Position of x (sec 3.2)
5:          k = [NF_θ(data[i])]
6:          Δ = pos − k
7:          Shift_Table[k].Δ = min(Shift_Table[k].Δ, Δ)
8:          Shift-Table[k].C += 1
9:      end for
10:     for k ← N − 1 ··· 0 do
11:         if Shift_Table[k].C = 0 then          ▷ Empty partitions
12:             Shift_Table[k].C = Shift_Table[k-1].C
13:             Shift_Table[k].Δ = Shift_Table[k-1].Δ + 1
14:         end if
15:     end for
16:     return Shift_Table
17: end procedure
```

3.4 Compressing the Shift-Table layer

Correcting the prediction of the model using the Shift-Table layer takes a single DRAM lookup irrespective of the size of the index. However, it might be of interest to reduce the size of the layer. The Shift-Table layer is an array of size N, containing <Δ, C> tuples. Further compression can be used to decrease the memory footprint of the Shift-Table layer.

One approach is to keep a single parameter instead of the <Δ, C> tuples. In this regard, a predicted position k should be mapped to the key that is in the median point among the keys in P_k, which is

$$\bar{\Delta}_k = \left[\Delta_k + \frac{C_k}{2}\right] \tag{3}$$

To correct using the $\bar{\Delta}_k$ values, the final position is computed as $pos = k + \bar{\Delta}_k$, which indicates where the search should be started without specifying the guaranteed range that should be searched. Therefore, search algorithms that require the boundaries specified such as binary search cannot be used for local search. As discussed in section 2.4, linear or exponential search can be used for local search without boundaries, but they are slightly slower if the error is considerable after the correction.

A second approach that complements the first one, is to shrink the size of the Shift-Table layer by merging nearby partitions. We can extend the definition of $\mathcal{P} = \{P_1, \cdots, P_N\}$ to allow partitions that have a size of $M < N$. We define M partitions $\mathcal{P}^M = \{P_1^M, \cdots, P_M^M\}$ where each partition is defined as:

$$P_k^M = \{x \mid [MF_\theta(x)] = k\} \tag{4}$$

Similarly, Δ_k^M is the minimum "move to the right" shifts that each of the keys in P_K^M need:

$$\Delta_k^M = \min\left(NF(x) - [NF_\theta(x)]\right) \quad \forall x \in P_k^M \tag{5}$$

and C_k should be defined such that the boundary is valid for all keys in P_K^M, which is:

$$C_k^M = \max(NF(x) - \underbrace{([NF_\theta(x)] + \Delta_k^M)}_{\text{start of the search window}}) \quad \forall x \in P_k^M \tag{6}$$

To combine approaches to compact the Shift-Table layer, we can use average drifts $\bar{\Delta}_k^M$ instead of the <Δ_k^M, C_k^M> pairs:

$$\bar{\Delta}_k^M = \left[\frac{1}{|P_k^M|} \sum_{x \in P_k^M} (NF(x) - [NF_\theta(x)])\right] \tag{7}$$

and then use $[NF_\theta(x)] + \bar{\Delta}^M_{[MF_\theta(x)]}$ as the corrected prediction. Suppose the same data as in Figure 5, but instead of a Shift-Table layer of size N, we use only M=30 partitions. Table 1 shows how a compact Shift-Table layer is built and used for correction, on a portion of the index. We use the same model ($F_\theta = [x/1000]$), hence the prediction is $NF_\theta(x) = [0.1x]$, and the partition corresponding to a key is $NF_\theta(x) = [0.03x]$. All of the records from data[35..39] are assigned to the same partition P_{23}^{30} and their predictions are shifted 40 records backwards. Note that when

$M \neq N$, a partition does not specify a single point (or range) for all of the keys in the partition. Instead, the position of a key after correction depends on both $NF_\theta(x)$ (prediction) and $MF_\theta(x)$ (partition number). For example, all keys belonging to P_{23}^{30}, i.e., data$[35 \cdots 39]$ have the same correction of $\bar{\Delta}_{23}^{30} = -40$, but their final predictions are different. Therefore, the correction error of a compact Shift-Table layer is less than the number of elements in the partitions.

Table 1: Illustration of Shift-Table with $M = 30$ mapping entries on an index with $N = 100$ keys

Index	34	35	36	37	38	39	40	41
key (x)	752	769	770	771	782	785	820	830
Predicted index= [0.1 x]	75	76	77	77	78	78	82	83
Error before correction	-41	-41	-41	-40	-40	-39	-42	-42
Partition (k) = [0.03 x]	22	23					24	
$\bar{\Delta}_k^{30}$	-41	-40					-42	
Prediction after correction	34	36	37	37	38	38	40	41
Error after correction	0	1	1	0	0	-1	0	0

The drift of P_k^M, namely $\bar{\Delta}_k^M$ is the index of the median key among the members of P_k^M. This means that if the key is predicted to be in the k'th partition (among the M partitions), the local search is done around $[NF_\theta(x)] + \bar{\Delta}_k^M$.

Using a Shift-Table layer of size $K < N$ does not affect the complexity of building the layer, which is $O(N) \times O(F_\theta) + O(M)$. However, if the midpoint-values are used (correction without specifying the boundary), it is possible to construct the map using a sample of the indexed keys, which comes at the cost of the accuracy. Using a sample of size $S < N$, the layer can be built in $O(S) \times O(F_\theta) + O(K)$ time.

Nonetheless, keep in mind that the Shift-Table layer is designed for applications that favour latency to memory footprint, hence reducing the memory footprint of the Shift-Table layer by a large factor will limit its margin for improvement as the fine-grained details of the empirical CDF will be lost to some extent.

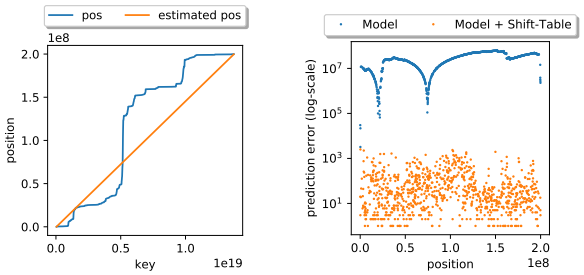
3.5 Measuring the error

Since the Shift-Table layer specifies a range for local search, the notion of error is not trivial. However, we can use the estimates without range ($\bar{\Delta}$), for which the correction picks the median value among the keys in the P_k. The error for the keys in each partition is $\left\{[\frac{C_k}{2}], \cdots, 0, \cdots, [\frac{C_k}{2}]\right\}$ if C_k is odd, and $\left\{[\frac{C_k}{2}] - 1, \cdots, 0, \cdots, [\frac{C_k}{2}]\right\}$ if C_k is even. The average error is approximately $C_k/4$.

In a learned index without Shift-Table, the error is the distance between $F(x)$ and $F_\theta(x)$. After correcting the model with the Shift-Table, however, the error only depends on the C_k values, i.e., a prediction error only occurs when $[F_\theta(x)]$ predicts the same position for multiple keys. Therefore, the local search range and the error are combinations of multiple step functions over the P_ks with $C_k > 1$.

The average error depends on the data distribution in the query workload. If the queries are uniformly sampled from the keys, then the average error is:

$$\bar{e} = \frac{1}{2N} \sum_{k \in \mathcal{P}} C_k^2 \tag{8}$$



(a) Example data & model **(b) Error**

Figure 6: Error correction using the Shift-Table layer

3.6 Behaviour of the Shift-Table layer

Figure 6 illustrates how the Shift-Table layer corrects the error of a linear interpolation model on the OSMC data. While the model is too simple to capture the patterns in data, the Shift-Table layer alone is effective for correcting the predictions. While the average error of the model is 28 million keys, Shift-Table reduces the error to only 129 keys.

Shift-Table corrects two types of error. First, when the model has a considerable local bias, which means that $NF(x)$ diverges significantly from $NF_\theta(x)$ in a sub-range of the data distribution. The second type of error is the fluctuations of the distribution between the nearby keys, for most of which the Shift-Table layer is very effective. The only type of error that can degrade the performance of the Shift-Table layer is when there is a congestion of keys in a small sub-range of values, leading to many of the keys being classified in a single layer, and hence having some partitions with high C_k.

The behavior of the Shift-Table layer and its error estimate indicates that it can be effective in eliminating different types of errors that models have. One common type of error is the local bias in the model, i.e., when the error of the model, i.e., $NF_\theta(x) - NF(x)$ has a considerable *bias* in some sub-ranges of the distribution, meaning that the F and F_θ diverge at some point. This happens when the model cannot capture the CDF in a local neighborhood. Table 2 shows that even if a single line is used as a model, which has a huge bias in most areas of the distribution, the Shift-Table layer can efficiently eliminate the huge bias of a fully linear model (a single line as a model), and reduces the error significantly such that the linear model outperforms all other algorithms for the real-world datasets, as well as the uspr dataset (sparse uniformly-distributed integers) which has a significantly higher variance than uniformly-distributed dense integers.

Another type of error that the Shift-Table layer eliminates is the local variance in the data, which is the fluctuations of the values between nearby keys. This type of error is very common in real-world data. For example, the *face, uspr, and uden* datasets all follow a uniform distribution, but they have different local variances, which is the amount of fluctuations in the nearby keys. The uden dataset is very easy to model using the learned indexes and does not require a helping layer such as Shift-Table. The other two datasets, however, are very hard to model using the learned index structures. This is because the Shift-Table model can easily correct the fluctuations of values (different increments between each two points), as long as the model does not predict a single record for a lot of nearby keys (resulting in a high C_k value).

3.7 Cost model of the Shift-Table layer

The accuracy of the model after correction with Shift-Table depends on the cardinalities of the partitions (C_i values). Ideally, if the records of each partition reside on a single cache line, the results will be retrieved in a single memory lookup. The cost of local search, i.e., the mapping between the accuracy in each partition and the latency to do local search depends on the hardware. As discussed in section 2.1, the latency of search for various ranges can be measured by a micro-benchmark over non-cached regions with different sizes. Let $L(s)$ be the measured latency of non-cached search over a range containing s records. The latency for looking up a key in a region of size s is $L(C_k)$. Assuming that the queries have the same distribution as the data points, the average lookup latency for the index is:

$$\text{Latency with Shift-Table} = \text{Latency}(F_\theta) + \frac{1}{N}\sum_{k\in\mathcal{P}} C_k L(C_k) \quad (9)$$

The cost model can also be used to estimate which of the local search algorithms should be used, by substituting in equation 9 the local search cost of each local search algorithm, i.e., $L(s)$ mappings for linear, binary, and exponential search; and for and their different implementations. Branch-optimized binary search would be the natural choice if the Shift-Table model can determine the boundary (if using the Δ_k, C_k pairs), otherwise either linear or exponential search should be chosen based on the latency estimate.

Taking the cost of running the Shift-Table layer into account, we should consider how much the correction improves the accuracy of the learned index model and hence estimate the speedup. The lookup time of the model without using the Shift-Table model can be estimated once the Shift-Table model is built, without running a speedup benchmark. The model error for each key is $\bar{\Delta}_k = \Delta_k + \frac{C_k}{2}$, therefore the estimated runtime of the index without correction is:

$$\text{Latency without Shift-Table} = \text{Latency}(F_\theta) + \frac{1}{N}\sum_{k\in\mathcal{P}} C_k L(\bar{\Delta}_k) \quad (10)$$

3.8 CDF model validity constraint

The correction layer requires the learned model to be a valid CDF function, i.e., $F_\theta(x)$ should be monotonically increasing: $x_i > x_j \longrightarrow F_\theta(x_i) >= F_\theta(x_j)$. Among our baselines, the RadixSplines learned index always produces a valid (increasing) CDF, but the RMI index does not always produce monotonically increasing predictions. In RMI, for example, the CDF model might decrease when using cubic models [30] or on the edge point between two models in the second-level. If $F_\theta(x)$ is not monotonically increasing, then the correction layer could identify a range that does not include the query result, because the values of x for which the learned model predicts k'th record are not in a contagious memory block.

A learned index model that is non-monotonic can still use the Shift-Table layer, as the output of the Shift-Table layer would still predict a position but it is not guaranteed that the position is in the predicted range. Therefore, the local search algorithm should check if the query is in the predicted range and perform a search outside of the range. Another hack for non-monotonic model is to use the $\bar{\Delta}$ midpoint-values instead of the Δ_k, C_k pairs, which predicts a location (instead of a range) to start the local search.

If the Shift-Table layer uses the $<\Delta_k^M, C_k^M>$ pairs, it can determine the range for local search and we can apply either linear or binary search, depending on the error range. We do linear search if the range is smaller than a threshold (8 keys, in our experiments), otherwise a binary search is performed. However, if it only contains the average shift values ($\bar{\Delta}_k$, it predicts a position without specifying the boundaries that contain the record; hence either linear or exponential search can be performed depending on the *average* error rate and performance objectives (average or worst-case latency).

3.9 Tuning the system

The Shift-Table layer is optional and adds overhead to the search. Therefore, enabling Shift-Table is only worthwhile if it can eventually accelerate the original learned index structure. An effective configuration of the index is a choice between 1) Using the model alone, 2) model + Shift-Table. Note that the Shift-Table layer is optional and can be deactivated with zero cost. The output of the model and the Shift-Table layer are of the same type and both represent a prediction of the records, hence if the Shift-Table layer is disabled, we can easily use the model alone for prediction of the records.

While tuning the system, the performance of each configuration can be directly measured using performance tests, or by measuring the model error and then using the cost model of the Shift-Table model on the bottom of the architecture (section 3.7).

The parameters of the architecture, i.e., the Shift-Table array size M and the parameters of the learned CDF model, can be tuned by computing the error estimate using Shift-Table's cost model, or alternatively, by running a performance tests on the built architecture. Our suggested default value for the Shift-Table layer is $M = N$, because using a mapping layer that has the same number of entries as the keys will ensures that the layer can exhibit its ultimate effect to eliminate the signed error, and does not have more latency compared to using smaller M values.

An advantage of Shift-Table is that the learned model does not need to be very accurate, as a correction will be applied anyway. Therefore, a more relaxed measure can be used instead of least-square error. In this paper, however, we do not learn the model w.r.t. the Shift-Table layer, for the sake of simplicity and to keep the Shift-Table layer detacheable (optional), preserving the assumption that the Shift-Table layer can be disabled to free up memory space on run-time while the model can still be used.

The accuracy of the learned model also determines the size of the entries of the Shift-Table layer. Each mapping entry should at most fit a Δ value of Δ_{MAX}, which is the maximum error of the model. If, for example, the error is smaller than $2^16/2$, then a 16-bit integer (`short` type) can be used.

4 EVALUATION

In this section, we compare the performance of our proposed method with the SOSD benchmark [1], which is a recent benchmark for search on sorted data. The benchmark includes learned indexes, classical indexes, and no-index search algorithms.

Experimental Setup. The algorithms are implemented in C++ and compiled with GCC 9.1. The experiments are performed on a system with 16 GB of memory and Intel Core i7-6700 (Skylake), which has four cores and is running at 3.4 GHz with 32 KB L1, 256 KB L2, and 8 MB L3 caches. The operating system is Ubuntu 18.04 with kernel version 4.15.0-65. In our setup, the LLC miss

[1] https://github.com/learnedsystems/SOSD/tree/mlforsys19

penalty measured by Intel Memory Latency Checker [2] is 36 ns, which is the minimum lookup time of an ideal index.

Note that all data resides in main memory. The range index finds the first indexed key that is equal to or bigger than the lookup key. Also, the keys on the physical layout are sorted (i.e., it is a clustered index), so that the entire result set of the range query can be returned once the first key is found. Similar to [22, 25], we only report the lookup time for the first result and do not include the scan times in our experiments because all indexes use the same layout for the data records.

Datasets. For the sake of reproducibility, we used the same datasets as in the SOSD benchmark, which contains four datasets synthetically generated from known distributions and four real-world ones. The synthetic data are generated from different distributions, namely *logn*: lognormal distribution $(0, 2)$, *norm*: normal distribution, *uden*: uniformly-generated dense integers, and *uspr*: uniformly-generated sparse integers. The real-world datasets are *face*: Facebook user IDs [42], *amzn*: book sale popularity from Amazon sales rank data [3], *osmc*: uniform sample of OpenStreetMap locations [4], and *wiki*: timestamps of edit actions on Wikipedia articles[5]. All datasets contain 200M unsigned integers.

Implementation details. Our experiments are based on the SOSD benchmark [22]. The baseline includes two learned indexes, namely RadixSpline [33] (RS), which uses linear splines; and Recursive Model Index (RMI), which uses a hierarchy of models. Note that RMI has a choice of different models and SOSD [22] specifically handpicked the best models for each of the datasets in the benchmark [6]. SOSD also includes no-index search algorithms such as binary search (BS), linear interpolation search (IS), and the recently suggested non-linear triple-point interpolation (TIP) [42]. We also compare against algorithmic index structures such as ART: Adaptive Radix Tree [26], FAST [21], RBS (Radix Binary Search): a two-stage algorithm in which a radix structure that maps a fixed-length key prefix to the range of all keys having that prefix and then a binary search is performed on the range [22], and STX implementation of B+tree [1]. Finally, we included four *On-the-fly* search algorithms, namely BS: Binary search (STL implementation), TIP: three-point interpolation [42], Interpolation search, which is similar to binary search but uses interpolated positions in each iteration, and IM: *Interpolation as a Model*: a dummy model that interpolates the key between the minimum and maximum value of the keys and then performs exponential search around the predicted key.

The experiments use either 32- or 64-bit unsigned integer IDs for the key (depending on the dataset), and 64-bytes for the payload.

4.1 The SOSD benchmark

To test the effectiveness of the suggested layers compared to learned indexes, we use a simple interpolation model (IM), i.e., $F_\theta(x) = (x - minVal)/(maxVal - minVal)$. Such a dummy model is deliberately chosen to purely delegate the burden of data modelling to the correction layers.

The Shift-Table layer has the same number of entries as the actual data, i.e., $M = N$. We followed the tuning procedure discussed in section 3.9: we start from the model (IM and RS) and consequently evaluate IM+Shift-Table and RS+Shift-Table. The cost of running the Shift-Table layer is around 40ns, which pays off by reducing the prediction error and thus lookup time. Therefore, based on the cost model of the Shift-Table layer (Section 3.7) and the error-to-latency micro-benchmark (Figure 2a), we should not add the Shift-Table layer if the error before adding the configuration is less than a threshold (10 records), or 2) the error of the index after adding the Shift-Table layer does not decrease by a factor of 10 (roughly equivalent to the 50-nanoseconds latency the additional layer, according to the error-to-latency micro-benchmark).

Table 2 compares the lookup times (nanoseconds per lookup) of the baseline algorithms with our dummy interpolation model (IM), and the two corrected versions, i.e., IM+Shift-Table and RS+Shift-Table. Note that ART does not support data with duplicate keys, and FAST does not support 64-bit keys. Also, interpolation search (IS) takes too much time on some datasets, because the execution time of interpolation search highly depends on the uniformity of data distribution, varying from O(loglogN) + O(1) iterations on uniform distributions, to O(N) iterations for very skew ones [42].

For the synthetic datasets, the difficulty of the datasets for our dummy linear interpolation model varies from very easy (uden64) to extremely hard (logn64). While the Shift-Table layer significantly improves a dummy layer on non-uniform data distributions, it cannot outperform the learned index models. This is not surprising, as all synthetic datasets (uniform, lognormal, and uniform) have a pattern derived from continuously differentiable density functions, hence the distribution is similar to a straight line on smaller sub-ranges as we "zoom in" the data distribution (e.g., see Figure 3c). Therefore, a learned index structure composed of linears at the bottom (including both RMI and RS) can effectively model the distribution using a very compact representation.

For the real-world data, however, the fluctuations in data severely affect both RMI and RS learned indexes. The Shift-Table layer, effectively corrects a highly inaccurate dummy IM model, such that it outperforms the RMI learned index by 1.5X to 2X on all datasets, while RS falls behind both. Keep in mind that RMI requires to be tuned with the best architecture and parameters, while Shift-Table does not require a manual training process and can even work with a simple model such as IM that is not trained, and yet deliver a lower latency.

Figure 7 shows the average build times of the indexes, along with the standard deviation bars indicating how the build time varies for different distributions. Please note that the RMI implementation used in the SOSD benchmark needs to be compiled for faster retrievals, however we did not include RMI's extra overhead for compiling the code and only reported the build time. IM+Shift-Table, the winner method latency-wise, also takes either the same or even less build time than the competing learned indexes.

4.2 Explaining the performance

The latencies reported in Table 2 present the fastest configuration for each learned index. In this section, we present the details of the tuning process to see the optimum performance of each learned index.

[2] https://software.intel.com/en-us/articles/intelr-memory-latency-checker
[3] https://www.kaggle.com/ucffool/amazon-sales-rank-data-for-print-and-kindle-books
[4] https://aws.amazon.com/public-datasets/osm
[5] https://dumps.wikimedia.org
[6] The architectures and parameters of the RMI models used for each dataset is specified at https://github.com/learnedsystems/SOSD/blob/mlforsys19/scripts/build_rmis.sh

Table 2: Comparison of lookup times (nanoseconds per lookup) with the SOSD benchmark. The red box indicates the base model (IM) and the enhanced versions.

		Algorithmic indexes				On-the-fly search				Learned indexes			
	Dataset	ART	FAST	RBS	B+tree	BS	TIP	IS	IM	IM + Shift-Table	RMI	RS	RS + Shift-Table
Synthetic	logn32	N/A	230	385	375	624	551	N/A	1384	166	**73.9**	83.9	143.5
	norm32	173	197	267	390	655	671	N/A	1479	88.2	**51.5**	60.3	96.4
	uden32	99.4	196	235	389	654	126	**32.3**	38.6	67.5	38.1	47.8	72.3
	uspr32	N/A	198	230	390	654	298	321	425	**89.7**	141	166	153.5
	logn64	238	N/A	622	427	674	377	N/A	1075	376	132	**109**	151.0
	norm64	214	N/A	317	427	672	705	N/A	1615	88.6	**51.7**	61.8	93.2
	uden64	104	N/A	255	428	670	142	**34.8**	40.4	67.4	39.8	47.9	71.8
	uspr64	216	N/A	244	427	673	329	338	**472**	**92.8**	145	182	154.6
Real-world	amzn32	N/A	208	243	393	658	569	3228	1524	**99.5**	185	236	110.8
	face32	179	203	238	388	654	717	792	861	**103**	213	310	142.8
	amzn64	N/A	N/A	284	428	676	578	3510	1575	**105**	189	238	119.3
	face64	290	N/A	257	427	671	925	1257	918	**149**	247	344	204.1
	osmc64	N/A	N/A	410	428	675	4617	N/A	1462	194	297	339	**177.2**
	wiki64	N/A	N/A	271	437	686	767	5867	1687	**94.2**	172	191	124.1

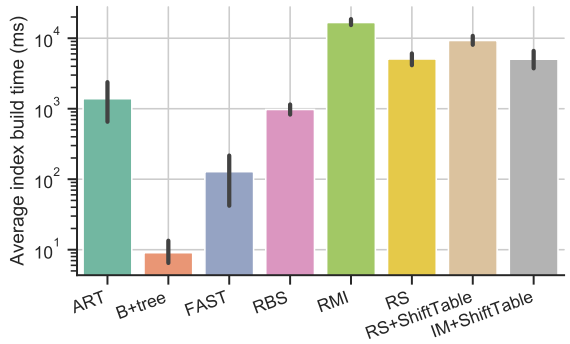


Figure 7: Build times (average time for all datasets)

For those indexes that have a parameter affecting the index size (such as the branching factor in B+tree, and the number of radix bits in ART, RS, and RBS), the performance can be tuned by evaluating the latency for different index sizes.

Figure 8 shows the latencies of the indexes for the face64 and osmc64 datasets, along with the average Log2 error, CPU instructions, and L1/LLC cache misses. IM+Shift-Table and RS+Shift-Table achieve faster lookup times on both datasets. For most indexes, except RMI and RBS, the latency does not improve beyond a certain optimum index size, after which the latency increases again. RBS has a much larger latency than both [IM/RS]-Shift-Table indexes of the same size, and extrapolating the RMI latencies also suggest that if we could extend RMI size to 1400MB (equal to Shift-Table's size), it could not achieve a game-changing performance on either of the datasets. Note that we could not run RMI with larger models because RMI embeds the parameters into the code, and the compile times for models larger than 400MB were astonishingly high.

Average Log2 errors indicate the average number of iterations in binary search for the last-mile search stage. Larger models result in lower Log2 errors in all indexes and lead to faster last-mile search, however, once the model exceeds the LLC cache sizes, cache-miss rate increases (when running the model), and hence the prediction time worsens. For RS, ART, and B+tree, the cache misses and extra overhead of running the models increases either the number of instruction, the cache misses, or both, enough to prevent the index from improving latency by increasing the footprint.

4.3 Layer size

As discussed in section 3.4, the Shift-Table layer can be compressed by merging multiple entries, hence reducing its footprint. Figure 9 shows the effect of the Shift-Table layer size on lookup time and prediction error. Shift-Table can operate in two modes: **R-1**: a full layer containing $<\Delta_k^M, C_k^M>$ pairs similar to Figure 5 that indicates the exact *range* for local search (hence enabling binary search); and **S-X**: a compressed single-entry map similar to Table 1 containing one $\bar{\Delta}_k^M$ entry per X records. Thus, S-X contains $M = N/X$ entries; and the memory footprint of S-1 is half the size of R-1.

The error of the S-1 Shift-Table is slightly more than that of R-1. This is due to the fact that S-1 is designed to draw boundaries for binary search; hence it always points to the first record of each partition; while R-1 always points to the middle of the partition and almost half the error of S-1. Performance-wise, however, S-1 always has the lowest latency, because its boundaries for the last-mile search operation do not need to be discovered using additional boundary-detection algorithms such as exponential search. As expected, compressing the Shift-Table by allocating one entry per X records increases the error and hence degrades

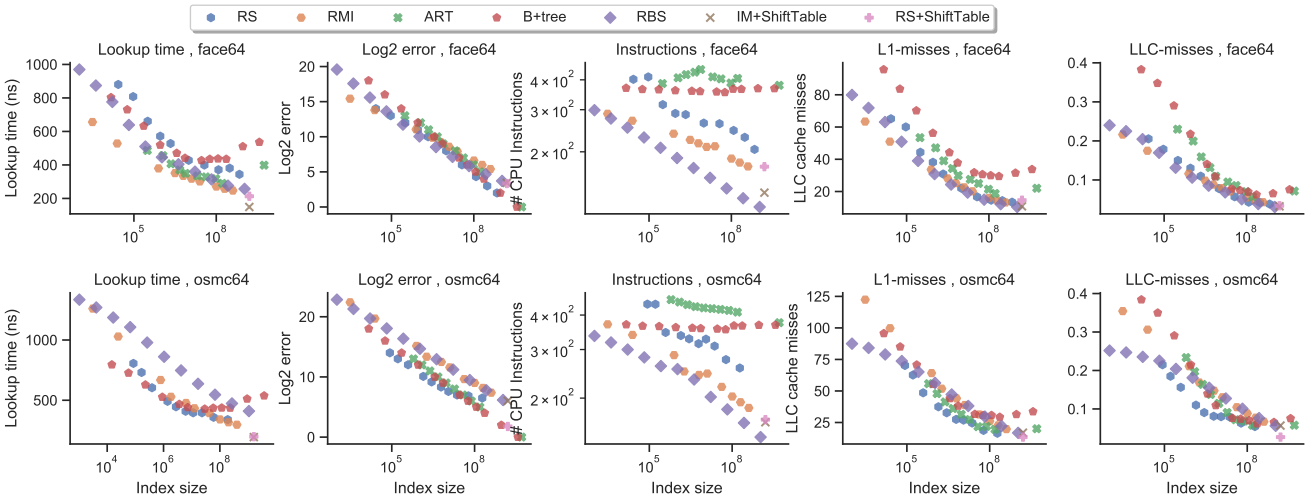


Figure 8: Analysis of the effect of index size on performance

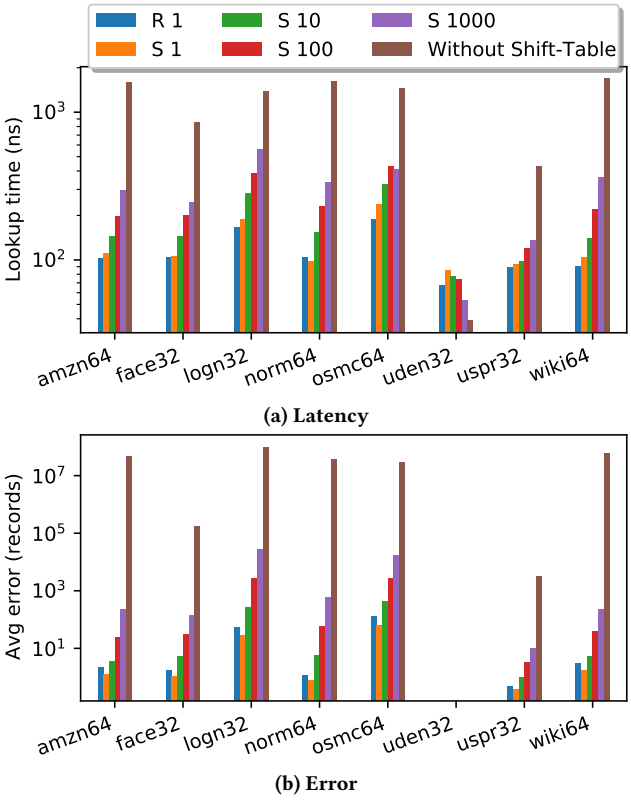


Figure 9: Analysis of the effect of Shift-Table layer size

the performance. This is due to the fact that with higher compression ratios, the ability of Shift-Table to "memorize" the fine-grained details of the data distribution degrades due to the loss of information after merging.

5 RELATED WORK

On-the-fly search on sorted data A fundamental problem that is studied for decades is how to find a key among a sorted list of items. The classic approach is binary search and numerous extensions have been suggested to improve it for special cases, most notably interpolation search [35] and exponential search [3]. For data distributions that are close to uniform, interpolation-search is shown to be very effective [13, 36, 42]. Due to the growing gap between CPU power and memory latency in the past decade, more advanced interpolation techniques such as three-point interpolation are becoming viable on modern hardware [42]. Exponential search enables binary search over an unbounded list. Exponential search is also extensively used in learned indexes when the key is more likely to be near a "guessed" location, but a guaranteed boundary around the guessed point that contains the data is not known [7, 25, 32].

Range indexes An alternative to on-the-fly binary search over sorted data is to keep the data in an index structure. Nonetheless, indexes that are built to answer range queries (such as B-trees) are similar to the binary search in that they need to keep the data sorted internally. Common index structures for range index include skiplists, B+trees, and radix-trees. The B+-tree is cache-efficient, but requires pointer chasing, which incurs multiple cache misses [14]. There has been a tremendous effort to make binary search trees and B+-trees efficient on modern hardware. For example, FAST [21] organizes tree elements efficiently to exploit modern hardware features such as the cache line and SIMD. Another common solution is to use compression techniques on the indexed keys, most notably as a radix-tree. Modern radix trees exploit hardware-efficient heuristics for fitting a distribution in memory (usually by building a heuristically-optimized compressed trie), such as adaptive radix index (ART) [5, 26], and Succinct Range Filter (SuRF) [44]. Skiplist is specifically efficient for concurrent updates workloads [41, 43].

Learned index structures Learned range indexes [7, 12, 25, 29, 33] have recently been suggested as an alternative to range indexes. In this approach, a model is trained from the data with the intent of capturing the data distribution and processing the queries more efficiently. We refer to the paper by Kraska et al. [25], which introduced the idea of the learned index. In a learned index, the CDF of the key distribution is learned by fitting a model, and the learned model is subsequently used as a replacement of the index (B+-trees or similar) for finding the location of the query results on the storage medium. Index learning frameworks such as the RMI model [25, 30] can learn arbitrary models [30], although a further theoretical study [9] as well as a recent experimental benchmark [22] have shown that simple

model like linear splines are very effective for datasets. Spline-based learned indexes include Piecewise Geometric Model index (PGM-index) [11], Fiting-tree [12], Model-Assisted B-tree (MAB-tree) [19], Radix-Spline [23], Interpolation-friendly B-tree (IF-Btree) [18] and some others [29, 40]. We refer to [10] for an extensive comparison of learned indexes. Recently, there has been numerous theoretical works [4, 27, 38, 39] on learned indexes. Also, numerous efforts have been made to handle practical challenges around using a learned index, including update-handling [7, 17] and designing a learned DBMS [24]. The idea of using a model of the data to boost an existing algorithmic index has been the center of focus in the past few years [14, 17, 19, 37]. In the multivariate area, learning from a sample workload has also shown interesting results [8, 20, 28, 32]. Aside from the main trend in learned indexes, which is on range indexing, machine learning has also inspired other indexing and retrieval tasks. This includes bloom filters [6, 31], multidimensional indexing on datasets with correlated attributes [15], and other applications [2, 16, 34].

6 CONCLUSION AND FUTURE WORK

Learning and modeling data distributions via machine learning approaches is a great idea for managing and analyzing data management systems. However, the approaches and objective functions that are common in machine learning problems are not necessarily optimal choices when the ultimate target is performance improvement. Instead of pushing machine learning model algorithm to its limits for highly accurate modeling of data distributions, it is more efficient if we only use ML models to approximate the high-level, generalizable "patterns" in data distribution (the holistic shape), and handle the fluctuations and fine-grained details of the distribution using a more hardware-efficient approach, outperforms learned models as well as algorithmic index structures even if a simple or somewhat dummy model such as min/max linear interpolation is used. The Shift-Table layer is effective in learning almost all distributions even without using models that require training from data, and takes only a single pass over the data points to build the layer. Our results show that even a simple linear model equipped with the Shift-Table enhancement layer outperforms trained and tuned learned indexes by 1.5X to 2X on real-world datasets.

Our current work only considers read-only workloads. We leave it as future work to adapt Shift-Table with workloads having updates. One idea is to capture the drifts in data distribution using update-tracking segments [17], and use Fenwick trees to estimate and correct the drifts in both the model and the Shift-Table.

REFERENCES

[1] STX. B+Tree C++ Template Classes. http://panthema.net/2007/stx-btree.
[2] Naiyong Ao, Fan Zhang, Di Wu, Douglas S Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. 2011. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *VLDB Endowment* 4, 8 (2011), 470–481.
[3] Jon Louis Bentley and Andrew Chi-Chih Yao. 1976. An almost optimal algorithm for unbounded searching. *Information processing letters* 5 (1976).
[4] Rasmus Bilgram and Per Hedegaard Nielsen. 2019. *Cost Models for Learned Index with Insertions.* Technical Report. University of Aalborg.
[5] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: a height optimized Trie index for main-memory database systems. In *SIGMOD*. 521–534.
[6] Zhenwei Dai and Anshumali Shrivastava. 2019. Adaptive learned Bloom filter (Ada-BF): Efficient utilization of the classifier. *arXiv:1910.09131* (2019).
[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*. 969–984.
[8] Mohamad Dolatshah, Ali Hadian, and Behrouz Minaei-Bidgoli. 2015. Ball*-tree: Efficient Spatial Indexing for Constrained Nearest-neighbor Search in Metric Spaces. arXiv:cs.DB/1511.00628
[9] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why are learned indexes so effective?. In *ICML*, Vol. 119. PMLR.
[10] Paolo Ferragina and Giorgio Vinciguerra. 2020. Learned data structures. *Recent Trends in Learning From Data* (2020), 5–41.
[11] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *VLDB Endowment* 13, 8 (2020), 1162–1175.
[12] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *SIGMOD*. 1189–1206.
[13] Goetz Graefe. 2006. B-tree indexes, interpolation search, and skew. In *DaMoN*.
[14] Goetz Graefe and Harumi Kuno. 2011. Modern B-tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.
[15] Ali Hadian, Behzad Ghaffari, Taiyi Wang, and Thomas Heinis. 2021. COAX: Correlation-Aware Indexing on Multidimensional Data with Soft Functional Dependencies. arXiv:cs.DB/2006.16393
[16] Ali Hadian and Thomas Heinis. 2018. Towards Batch-Processing on Cold Storage Devices. In *ICDEW*.
[17] Ali Hadian and Thomas Heinis. 2019. Considerations for handling updates in learned index structures. In *AIDM*.
[18] Ali Hadian and Thomas Heinis. 2019. Interpolation-friendly B-trees: Bridging the Gap Between Algorithmic and Learned Indexes. In *EDBT*.
[19] Ali Hadian and Thomas Heinis. 2020. MADEX: Learning-augmented Algorithmic Index Structures. In *AIDB*.
[20] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures. In *AIDB*.
[21] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*. 339–350.
[22] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).
[23] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *AIDM*.
[24] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *CIDR*.
[25] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD*. 489–504.
[26] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49.
[27] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. 2019. A Scalable Learned Index Scheme in Storage Systems. *arXiv:1905.06256* (2019).
[28] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD*.
[29] Anisa Llavesh, Utku Sirin, Robert West, and Anastasia Ailamaki. 2019. Accelerating B+tree Search by Using Simple Machine Learning Techniques. In *AIDB*.
[30] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *SIGMOD*. 2789–2792.
[31] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Related Structures. *arXiv:1802.00884* (2018).
[32] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-dimensional Indexes. In *SIGMOD*. 985–1000.
[33] Thomas Neumann and Sebastian Michel. 2008. Smooth interpolating histograms with error guarantees. In *BNCOD*. Springer, 126–138.
[34] Harrie Oosterhuis, J Shane Culpepper, and Maarten de Rijke. 2018. The potential of learned index structures for index compression. In *ADCS*.
[35] W Wesley Peterson. 1957. Addressing for random-access storage. *IBM journal of Research and Development* 1, 2 (1957), 130–146.
[36] CE Price. 1971. Table lookup techniques. *Comput. Surveys* 3, 2 (1971), 49–64.
[37] Wenwen Qu, Xiaoling Wang, Jingdong Li, and Xin Li. 2019. Hybrid Indexes by Exploring Traditional B-Tree and Linear Regression. In *WEBIST*. 601–613.
[38] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. 2018. Deja Vu: an empirical evaluation of the memorization properties of ConvNets. *arXiv:1809.06396* (2018).
[39] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. 2019. Spreading vectors for similarity search. In *ICLR*.
[40] Naufal Fikri Setiawan, Benjamin IP Rubinstein, and Renata Borovica-Gajic. 2020. Function Interpolation for Learned Index Structures. In *ADC*. 68–80.
[41] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. 2016. Cache-sensitive skip list: Efficient range queries on modern cpus. In *DaMoN*. Springer, 1–17.
[42] Peter Van Sandt, Yannis Chronis, and Jignesh M Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *SIGMOD*. 36–53.
[43] Zhongle Xie, Qingchao Cai, HV Jagadish, Beng Chin Ooi, and Weng-Fai Wong. 2017. Parallelizing skip lists for in-memory multi-core database systems. In *ICDE*. IEEE, 119–122.
[44] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. Surf: Practical range query filtering with fast succinct tries. In *SIGMOD*. 323–336.

An Efficient and Secure Location-based Alert Protocol using Searchable Encryption and Huffman Codes

Sina Shaham
University of Southern California
Los Angeles, USA
sshaham@usc.edu

Gabriel Ghinita
University of Massachusetts
Boston, USA
gghinita@cs.umb.edu

Cyrus Shahabi
University of Southern California
Los Angeles, USA
shahabi@usc.edu

ABSTRACT

Location data are widely used in mobile apps, ranging from location-based recommendations, to social media and navigation. A specific type of interaction is that of *location-based alerts*, where mobile users subscribe to a service provider (SP) in order to be notified when a certain event occurs nearby. Consider, for instance, the ongoing COVID-19 pandemic, where contact tracing has been singled out as an effective means to control the virus spread. Users wish to be notified if they came in proximity to an infected individual. However, serious privacy concerns arise if the users share their location history with the SP in plaintext.

To address privacy, recent work proposed several protocols that can securely implement location-based alerts. The users upload their encrypted locations to the SP, and the evaluation of location predicates is done directly on ciphertexts. When a certain individual is reported as infected, all matching ciphertexts are found (e.g., according to a predicate such as "10 feet proximity to any of the locations visited by the infected patient in the last week"), and the corresponding users notified. However, there are significant performance issues associated with existing protocols. The underlying searchable encryption primitives required to perform the matching on ciphertexts are expensive, and without a proper encoding of locations and search predicates, the performance can degrade a lot. In this paper, we propose a novel method for variable-length location encoding based on Huffman codes. By controlling the length required to represent encrypted locations and the corresponding matching predicates, we are able to significantly speed up performance. We provide a theoretical analysis of the gain achieved by using Huffman codes, and we show through extensive experiments that the improvement compared with fixed-length encoding methods is substantial.

1 INTRODUCTION

Location-based alerts are an emerging area of mobile apps that are very relevant to domains such as public safety, healthcare and transportation. For instance, users may want to subscribe to services that notify them whether an imminent danger exists in their close proximity (e.g., an active shooter situation). Or, in the recent context of COVID-19, mobile users wish to be notified if they came in close proximity to an individual who was diagnosed with the disease. While the advantages of location-based alerts are undeniable, they also introduce serious privacy concerns: in order to benefit from such services, users periodically upload their locations to a service provider (SP). The SP monitors large number of individuals, and evaluates spatial predicates to determine which individuals should be alerted. Disclosing individual locations can leak sensitive personal details to the SP, which may in turn share the data with third parties. And even in cases where the SP is fully trusted, it can be subject to cyber-attacks, or subpoenas by governments, resulting in the users' moving history being exposed.

Movement data can disclose sensitive details about an individual's health status, political orientation, alternative lifestyles, etc. Therefore, it is crucial to support location-based alerts while at the same time protecting user privacy. This problem has been recently studied in literature, and formulated in the context of *secure alert zones* [14, 21, 23], where users report their *encrypted* locations to the SP, and the SP evaluates alert predicates on encrypted data. These approaches require a special kind of encryption that allows predicate evaluation on ciphertexts, namely *searchable encryption (SE)* [5, 19, 24]. However, the SE primitives are not specifically designed for geospatial queries, but rather for arbitrary keyword or wildcard queries. As a result, a data mapping step must transform spatial queries to the primitive operations supported on ciphertexts. Due to this translation, the performance overhead can be significant.

Some solutions use *Symmetric Searchable Encryption (SSE)* [11, 19, 24], where a trusted entity knows the secret key of the transformation, and collects the location of all users before encrypting them and sending the ciphertext to the service provider. While the performance of SSE can be quite good, the system model that requires mobile users to share their cleartext locations with a trusted service is not adequate from a privacy perspective, since it still incurs a significant amount of disclosure.

Prior work in secure alert zones [14, 21, 23] uses *Hidden Vector Encryption (HVE)* [5], which is an *asymmetric* type of encryption that allows direct evaluation of predicates on top of ciphertexts. Each user encrypts her own location using the *public* key of the transformation, and no trusted component that accesses locations in clear is required. However, the performance overhead of HVE in the spatial domain remains high.

In existing HVE work for geospatial data [14], [21], the data domain is partitioned into a hierarchical data structure, and each node in this structure is assigned a binary string identifier. The binary representation of each node plays an important part in query encoding, and it influences the amount of computation that needs to be executed when evaluating predicates on ciphertexts. In [14], the earliest solution for secure alert zones, the impact of the specific encoding is not evaluated in-depth. In [23], the geospatial data domain is embedded to a high-dimensional hypercube, and then graph embedding [7] is applied to reduce the computation overhead in the predicate evaluation step.

However, all previous solutions use fixed-length encoding of locations and alert zones, meaning that the same number of bits is used to represent each location. In cases where the distribution of alert zones and/or locations is not uniform, using fixed-length encoding can introduce unnecessary overhead. Motivated by this fact, we propose techniques to reduce the computational overhead of HVE by using variable-length encoding. Specifically,

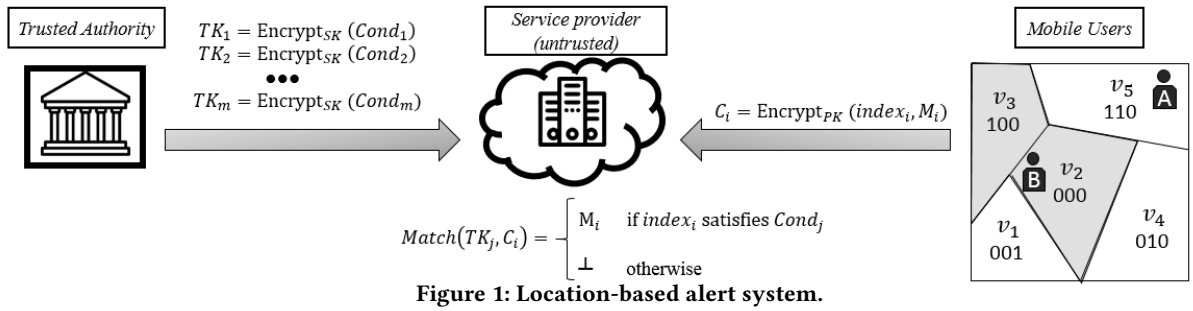
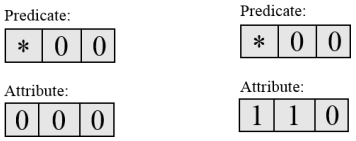


Figure 1: Location-based alert system.



(a) Match **(b) Nonmatch**

Figure 2: HVE evaluation

we use Huffman compression codes to represent both user locations and alert predicates. Areas of the domain that are more popular, or more likely to result in a secure alert being triggered, are encoded with fewer bits than less popular areas. This allows us to perform spatial query execution on ciphertexts in a less computationally-intensive manner.

Our specific contributions are:

- We consider for the first time the use of variable-length encoding, specifically Huffman compression codes, for the problem of secure alert zones on encrypted location data;
- We devise specialized domain encoding techniques for both user locations and alert zones that take into account location popularity;
- We provide algorithms to evaluate the secure alert zone enclosure predicates directly on ciphertexts when both user locations and alert zones are represented using variable-length encoding;
- We perform an extensive experimental evaluation which shows that the proposed approach reduces considerably the performance overhead of secure alert zones compared to fixed-length encoding approaches.

The rest of the paper is organized as follows: Section 2 introduces necessary background and the system model. Section 3 provides the details of the proposed variable-length encoding techniques for user locations and alert zones. Section 4 generalizes our solution to non-binary identifiers. Section 5 analyzes the overhead of variable-length encoding on ciphertext size. Section 6 provides a security discussion, followed by evaluation of the proposed approach on both real and synthetic datasets in Section 7. We survey related work in Section 8 and conclude in Section 9.

2 BACKGROUND

Consider a map divided into a set of n non-overlapping partitions

$$\mathcal{V} = \{v_1, , v_2, ..., v_n\}. \quad (1)$$

Each partition v_i represents a spatial area on the map referred to as *cell*. Cells are identified by a unique binary code called *index*, and can have arbitrary shapes and sizes (although equal-size square cells are most likely in practice). We refer to the partitioning as a *grid*. The assignment of indexes to cells is referred to as *grid encoding*. All indexes must have the same length for security purposes (to prevent an adversary from distinguishing cells based on length). Fig. 1 shows a sample grid with five cells, each associated with an index of length three.

When an event of interest occurs, an *alert zone* is created, which spans a number of grid cells. We refer to such cells interchangeably as *alert cells* or *alerted cells*. In Fig. 1, cells v_3 and v_2 associated with the indexes 100 and 000 (shown highlighted) are alert cells. We denote the likelihood of cell v_i being alerted by $p(v_i)$, or alternatively p_i. Our goal is to exploit alert cell likelihoods in order to choose an encoding that reduces the computational complexity of HVE.

2.1 Hidden Vector Encryption

Hidden Vector Encryption (HVE) [5] is a searchable encryption system that supports predicates in the form of conjunctive equality, range and subset queries. Search on ciphertexts can be performed with respect to a number of *index attributes*. HVE represents an attribute as a bit vector (each element has value 0 or 1), and the search predicate as a *pattern* vector where each element can be 0, 1 or '*' that signifies a wildcard (or "don't care") value. Let l denote the HVE *width*, which is the bit length of the attribute, and consequently that of the search predicate. A predicate evaluates to $True$ for a ciphertext C if the attribute vector I used to encrypt C has the same values as the pattern vector of the predicate in all positions that are not '*' in the latter. Fig. 2 illustrates the two cases of *Match* and *Non-Match* for HVE.

HVE is built on top of a symmetrical bilinear map of composite order [5], which is a function $e : \mathbb{G}\times\mathbb{G} \rightarrow \mathbb{G}_T$ such that $\forall a, b \in G$ and $\forall u, v \in \mathbb{Z}$ it holds that $e(a^u, b^v) = e(a, b)^{uv}$. $\mathbb{G}$ and $\mathbb{G}_T$ are cyclic multiplicative groups of composite order $N = P \cdot Q$ where P and Q are large primes of equal bit length. We denote by $\mathbb{G}_p$, $\mathbb{G}_q$ the subgroups of $\mathbb{G}$ of orders P and Q, respectively. Let l denote the HVE *width*, which is the bit length of the attribute, and consequently that of the search predicate. HVE consists of the following phases:

Setup. The public/private (PK/SK) key pair has the form:

$$SK = (g_q \in \mathbb{G}_q, \quad a \in \mathbb{Z}_p, \quad \forall i \in [1..l] : u_i, h_i, w_i, g, v \in \mathbb{G}_p)$$

To generate PK, we first choose at random elements $R_{u,i}, R_{h,i}$, $R_{w,i} \in \mathbb{G}_q, \forall i \in [1..l]$ and $R_v \in \mathbb{G}_q$. Next, PK is determined as:

$$PK = (g_q, \quad V = vR_v, \quad A = e(g, v)^a,$$

$$\forall i \in [1..l] : U_i = u_i R_{u,i}, \quad H_i = h_i R_{h,i}, \quad W_i = w_i R_{w,i})$$

Encryption uses PK and takes as parameters index attribute I and message $M \in \mathbb{G}_T$. The following random elements are generated: $Z, Z_{i,1}, Z_{i,2} \in \mathbb{G}_q$ and $s \in \mathbb{Z}_n$. Then, the ciphertext is:

$$C = (C^{'} = MA^s, \quad C_0 = V^s Z,$$

$$\forall i \in [1..l] : C_{i,1} = (U_i^{I_i} H_i)^s Z_{i,1}, \quad C_{i,2} = W_i^s Z_{i,2})$$

Token Generation. Using SK, and given a search predicate encoded as pattern vector I_*, a search token TK is generated

as follows: let J be the set of all indexes i where $I_*[i] \neq *$. We randomly generate $r_{i,1}$ and $r_{i,2} \in \mathbb{Z}_p, \forall i \in J$. Then

$$TK = (I_*, K_0 = g^a \prod_{i \in J} (u_i^{I_*[i]} h_i)^{r_{i,1}} w_i^{r_{i,2}},$$

$$\forall i \in [1..l] : K_{i,1} = v^{r_{i,1}}, \quad K_{i,2} = v^{r_i,2})$$

Query is executed at the service provider, and evaluates if the predicate represented by TK holds for ciphertext C. The server attempts to determine the value of M as

$$M = C^{'} / (e(C_0, K_0) / \prod_{i \in J} e(C_{i,1}, K_{i,1}) e(C_{i,2}, K_{i,2}) \tag{2}$$

If the index I based on which C was computed satisfies TK, then the actual value of M is returned, otherwise a special number which is not in the valid message domain (denoted by $\perp$) is obtained.

The HVE query, or matching, is the most important operation in a location-based alert system, because it is executed every time an alert occurs, and it requires processing of a large number of ciphertexts. *Our goal is to reduce the overhead of matching, and the most direct way to do so is by reducing the number of non-star bits in a token, since the number of expensive bilinear maps is proportional to the count of non-star bits.*

2.2 System Model

The architecture of location-based alert systems is shown in Fig. 1. There are three types of entities: mobile users, a service provider (SP) and a trusted authority (TA).

Mobile users subscribe to the location-based alert system and periodically submit their encrypted location updates. Users want to be notified when they are in an alert cell, without their privacy being compromised. They the public key (PK) of the HVE cryptosystem to encrypt their locations before sending them to the SP. For example, users A and B on the grid encrypt their indexes 110 and 000, generating two ciphertexts C_A and C_B, respectively.

The **Trusted Authority (TA)** has the secret key (SK) of the HVE cryptosystem. In practice, the TA role could be played by a reputable organization such as a law enforcement agency, or the center for disease control, who issue *HVE search tokens* corresponding to alerts. The TA does *not* have access to user locations, and is assumed not to collude with the SP. The TA is acting in the interest of the general public, but does not have the infrastructure to run a complex alert system, which is why this service is outsourced to the SP. One important aspect when generating tokens is to minimize the number of non-star bits in a token, in order to reduce the computational overhead of matching. A common approach is to use binary minimization on the cell identifiers. For example, the two alerted indexes 100 and 000 are combined using binary expression minimization to obtain *00, then, the new index is encrypted using the SK to create a token with two non-star bits, instead of two tokens with three non-star bits each. The overhead is reduced from six sets of bilinear pairings to two.

The SP implements the alert service. It receives encrypted updates from users and tokens from the TA, and performs the matching to decide whether encrypted location C_i of user i falls within alert zone j represented by token TK_j. If the *matching* outcome is positive, the SP learns that the user is inside the alert zone, and notifies the user. For a matching process to result in a positive outcome, all the token's non-star bits should exactly match the user index. Star bits ('don't care' bits), as the name suggests, match with either a zero or one bit in the user index. Note that all received information from users and the TA is encrypted in the matching process, and the search happens over encrypted data only.

Table 1: Summary of notations.

Symbol	Description
n	Number of cells
$\mathcal{V} = \{\bigcup v_i\}$	Set of all cells
$p(v_i)$	Probability of cell v_i becoming alerted
C_j	Encrypted location of user j
TK_j	Token j
M_j	Message of user j
RL	Depth of prefix tree (reference length)
r_i	ith internal node of tree
Pois(λ)	Poisson distribution; occurrence rate λ
Σ	Identifier symbol alphabet
γ	*Euler-Mascheroni* constant
ϕ	Golden ratio
$a[i:j]$	Returns elements i to $j-1$ of array a
$\overline{x_1 x_2 ... x_l}$	Concatenation of symbols x_1 to x_l

Revisiting the example in Fig. 1, the outcome of matching between token *00 and user B's ciphertext corresponding to index 000 is positive (all the non-star bits match); however, the matching outcome between *00 and 110 (user A) is negative as the second bits do not match. From the mathematical derivation of HVE (2.1), the HVE system's computation complexity is proportional to the number of non-star bits in the tokens. Therefore, a good grid encoding reduces the overall number of non-star bits in tokens to minimize the HVE computational overhead.

2.3 Motivation and Scope

While prior work made important steps toward secure and scalable location-based alert systems, important performance issues still need to be addressed. The pioneering work in [14] was the first to use searchable HVE encryption in the context of locations, but assumed that all data domain regions are equally likely to be part of an alert zone. Later in [23], it was shown that if there are significant differences in likelihood of distinct regions to be part of an alert zone, then performance can be significantly boosted. However, both [14] and [23] use fixed-length encoding, i.e., the same number of bits are used to represent each cell. Hence, their performance overhead depends entirely on their ability to aggregate search tokens. When alert zones consist of a relatively large number of co-located alert cells, fixed-length encoding methods are able to perform effectively binary minimization of identifiers, and reduce overhead. This may be sufficient in some scenarios such as an active shooter, or a gas leak, where there is an epicenter of the event, and a range around the epicenter (often circular) within which users must be alerted. The range can be large, for instance in the order of hundreds of meters.

However, in other applications, alert zones may be compact and sparse. For instance, consider the case of contact tracing – an important task in controlling pandemics, such as COVID-19. In this case, there will be a number of distinct alert zones, corresponding to the set of locations visited by a COVID-19 patient. For each individual site, the range of the query is relatively small, for instance, several meters around the patient location for direct spread. Or, in the case of surface spread or aerosol transmission, the query may be restricted to a room, or a store, which may be in the order of $10-20$ meters in size. There are insufficient cells in the alert zones to allow for effective token aggregation with fixed encoding, and the performance obtained may be poor.

Our goal is to address this latter case, and we do so by using a novel variable-length encoding approach. In this case, it is important to use fewer representation bits for high-probability regions. While our advantage is greatest for small, sparse alert zones, we show in our empirical evaluation in Section 7 that variable-length encoding can outperform fixed-length approaches for a wide choice of alert zone sizes, and mixed-size workloads.

Normalizing the cell probability values over the domain space reveals how likely a cell is to be alerted compared to others. A typical stochastic distribution used to model sporadic events is *Poisson distribution*, characterized as follows.

Theorem 1. *If a random variable Y represents the number of alert cells on the grid, then, it approximately follows Poisson distribution (Pois(λ)) with the occurrence rate of one (λ = 1).*

Proof. An alert zone event on the map is a subset of cells $v_1, , v_2, ..., v_n$, where n is a large value and each probability $p(v_i)$ is relatively small. Moreover, the events are either independent or weakly dependent of each other. Let

$$Y = \sum_{i=1}^{n} I(v_i) \tag{3}$$

count how many of the cells are alerted, in which I is an indicator random variable having a value of one when the cell is alerted and zero otherwise. Based on the Poisson distribution, the random variable Y can be approximated with rate $\lambda = \sum_{i=1}^{n} p(v_i) = 1$. Therefore, the probability of having k alert cells is given by

$$p(Y = k) = \frac{e^{-1}}{k!}. \tag{4}$$

□

One can see from the Poisson distribution that the likelihood of having a large number of alert cells is low. The maximum probability corresponds to having only a single alert cell in a zone, and then it drops significantly. This motivates our technique for dealing effectively with cases where alert zones are compact.

3 LOCATION-BASED ALERTS WITH VARIABLE-LENGTH ENCODING

In Section 3.1 we provide an overview of Huffman codes; Section 3.2 presents the proposed location encoding scheme; Section 3.3 introduces the token minimization process.

3.1 Prefix and Huffman Codes

Generally, any uniquely-decodable representation used to transmit information is a *prefix code*, i.e., it follows the *prefix property*, which requires that no whole code can be part of any other code. For example, [000, 001, 01, 10, 11] is a prefix code as no code starts with any other code in the set. A well-known theorem based on *Kraft inequality* [10] states that any prefix over an alphabet of size two with string lengths of l_1 to l_n must satisfy the inequality

$$\sum_{i=1}^{n} \frac{1}{2^{l_i}} - 1 \leq 0, \tag{5}$$

and conversely, given a set of string lengths that satisfies the Kraft inequality, there exists a prefix code with these string lengths. Let the tuple $\mathcal{P} = (p_1, p_2, ..., p_n)$ defined over space partitioning $\mathcal{V}$ indicate the likelihood of cells $v_1, \cdots v_n$ becoming alert cells. Furthermore, suppose that the function $f(l_1, l_2, ..., l_n)$ returns the average symbol length with no minimization, and $f_M(l_1, l_2, ..., l_n)$ returns the average reduction in number of bits in the minimization process. Given the tuple of cells and probabilities, the objective of a minimal encoding is to generate a prefix code $C(\mathcal{P}) = (c_1, c_2, ..., c_n)$ as follows:

$$\begin{aligned} &\text{minimize} \quad L(C(\mathcal{P})) = \sum_{i=1}^{n} p(v_i) \times length(c_i) \\ &\text{subject to} \quad L(C(\mathcal{P})) \leq L(\mathcal{T}(\mathcal{P})) \text{ for any code } \mathcal{T}(\mathcal{P}) \end{aligned}$$

Note that f_M, which indicates the amount of minimization, is not necessarily a function. For example, a previously used minimization approach based on Karnaugh maps [14] does not always result in a unique output. The NP-hardness of the above problem based on fixed-length codes is shown in [23].

The most well-known prefix code is the *Huffman encoding*, widely used in communication systems as it results in optimal decodable average code length. The main idea behind Huffman codes is that more common symbols are represented with fewer bits compared with the less common symbols. In grid encoding, it is desirable to encode symbols that have higher probabilities of being in alert zones with fewer bits than the less likely ones. Given the tuple of cells and probabilities, the objective of Huffman encoding is to generate a prefix code that minimizes the average length of codewords:

$$\begin{aligned} &\text{minimize} \quad f(l_1, l_2, ..., l_n) - f_M(l_1, l_2, ..., l_n) \\ &\text{subject to} \quad \sum_{i=1}^{n} \frac{1}{2^{l_i}} - 1 \leq 0 \\ &\qquad\qquad l_i > 0, \quad \forall i = 1, .., n \end{aligned} \tag{6}$$

Prefix Trees. An intuitive way to discover whether the prefix property holds for a code is to draw its associated binary tree, called prefix tree. The prefix tree is constructed by assigning an empty character to the root and descending through the tree. At each branching point, we either choose to go left by adding a zero character or move to the right child by adding a character '1' to the root string. We call the tree's depth *reference length (RL)*. This number also indicates the maximum length of a prefix code. Moreover, the subtree roots are referred to as *interior nodes* of the prefix tree, and the leaf nodes are the *prefix codes*. Fig. 4b, shows a prefix tree with an RL of three. As an example, the prefix code '001' is generated by traversing nodes r_4, r_2, and r_1.

3.2 Proposed Coding Scheme

The focus of prior work on secure alert zones [14, 23] has been on fixed-length codes. Such codes are indeed a special case of prefix codes, in which the tree is balanced, and no assigned code can start with another. Next, we show how variable-length codes can be used in conjunction with HVE. An overview of the proposed approach is presented in Fig. 3. Based on a given prefix code, the TA generates grid indexes where each index is a unique identifier of a cell in the grid. In addition to grid indexes, a coding tree is generated for the purpose of token minimization. Given the set of indexes associated with the alert cells, the TA applies the proposed minimization algorithm and transmits the encrypted tokens to the SP. Fig. 4 serves as a running example.

Our approach consists of four steps:

I. Generation of Probabilities: Our coding scheme relies on a set of probabilities for each cell of the location domain to be part of an alert zone. This step is a prerequisite to our approach, and thus performed independently of the encoding. Such probabilities are application dependent, and can be generated based on a trained

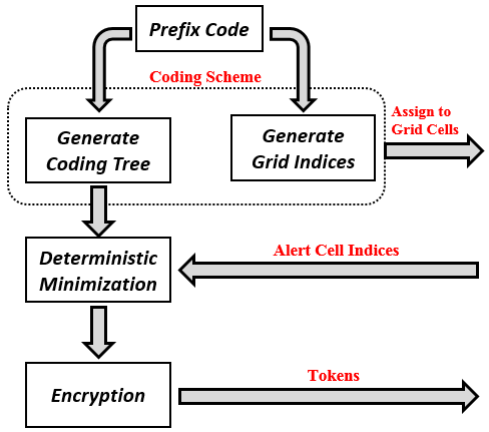


Figure 3: Overview of HVE with variable-length codes.

machine learning model. In the example of Fig. 4a, we have five cells $\mathcal{V} = (v_1, v_2, v_3\, v_4, v_5)$ with alert probabilities of

$$\mathcal{P}=(p(v_1)=0.1,\ p(v_2)=0.2,\ p(v_3)=0.5,\ p(v_4)=0.4,\ p(v_5)=0.6).$$

For grids entailing a high correlation between alert probabilities of cells, the setting in [23] or deep learning models such as [2] can be used to find the stationary distribution of probabilities, leading to a more accurate probabilistic model.

II. Prefix tree : An arbitrary prefix code defined over alphabet $\Sigma = \{0, 1\}$ can be represented by a binary tree with the prefix codes located on the leaves of the tree. We are not just interested in the generated prefix codes, but also in the codes associated with the internal nodes of the tree. Therefore, internal nodes are also stored as well as the generated prefix codes.

The topology of the tree is stored by recording five attributes of each node: left child (leftChild), right child (rightChild), parent node (parentNode), weight, and the associated code. The weight of a node represents its frequency. The leaf nodes have a frequency equal to their probability, and the weight of a parent node is found by the addition of its immediate children's weights (i.e., *Huffman mechanism*). The prefix tree is not used directly in the prefix coding scheme, but two sets of codes are generated based on the prefix tree; one is used for identifying grid cells referred to as cell *indexes*, and another is used by the TA to perform token minimization. Once the base codes are assigned for each node of the tree, two sets of *padding* are conducted, one for indexes assigned to the cells, and one used as a guideline for the token generation. The padding leads to a length of RL (i.e., equal length) for all codewords and indexes. Recall that equal ciphertext lengths is a requirement for security. However, the variable-length codes affect the ciphetexts and token contents in a way that allows fast processing. Furthermore, the padding prevents an adversary from distinguishing among ciphertexts.

III. *Grid indexes:* the prefix codes (leaves on the prefix tree) are padded from the right-hand side with zeros if they have a length less than RL. In our example, the generated prefix codes are $\{v_1 : 001, v_2 : 000, v_3 : 10, v_4 : 01, v_5 : 11\}$ which are transformed to $\{v_1 : 001, v_2 : 000, v_3 : 100, v_4 : 010, v_5 : 110\}$ after padding with zeros. We refer to zero-padded prefix codes as *indexes*. Once codes are created, they are assigned to corresponding cells identified by their probabilities. The assigned indexes to the sample grid are shown in Fig 4c. These are the indexes utilized by users to identify the cell they are enclosed by.

Algorithm 1 Coding Scheme

Input: Root; $\mathcal{V}$;

```
1:  //Root traversal to generate codes
2:  function TRAVERSE(Root)
3:      if Root has no children then
4:          return True
5:      else
6:          Root.leftChild = Root.code + '0'
7:          Root.rightChild = Root.code + '1'
8:          Traverse(leftChild)
9:          Traverse(rightChild)
10:
11: Traverse (Root)
12: //Generate indexes assigned to cells
13: RL ← depth of tree
14: for all leaf nodes do
15:     index = node.code
16:     while len(node.code) < RL do
17:         index = index +' 0'
18:     Assign index to v_i that has p(v_i) = node.weight
19:
20: //Generate coding tree
21: for all nodes do
22:     while len(node.code) < RL do
23:         node.code = node.code + '*'
24: //codingTree is the set of all nodes, alternatively Root can be
    returned
25: return codingTree
```

IV. *Coding tree:* the coding tree is used by the trusted authority to generate tokens. The coding tree is constructed by adding star bits on the right side of the prefix codes as well as the internal nodes on the prefix tree if they have a length less than RL. The padding for the sample grid is shown in Fig. 4d. The codes on the coding tree are referred to as *codewords*.

Algorithm 1 formally presents how indexes and the coding tree are generated for a given prefix tree. The inputs to the algorithm are the tree root, grid cells, and their probabilities. The tree root is sufficient for reconstructing the tree as children and parents are presumed to be recorded. The algorithm traverses through nodes to generate the prefix tree. Next, indexes of the grid are generated and assigned to the grid cells, and finally, the coding tree is completed and returned as the output of the algorithm.

Algorithm 2 presents how the Huffman tree is generated. The algorithm starts by creating a node (leaf node) for each cell of the grid, sorting them in ascending order based on their weights, and placing them in a priority queue. Recall that the weights of the leaf nodes are the probability of cells becoming alerted. Next, while the length of the queue is greater than one, the algorithm extracts two nodes with the minimum weights and creates a new internal node (newNode) with a weight equal to the addition of two extracted nodes. The new node is assigned as the parent of extracted nodes, and the extracted nodes are assigned as left and right children of the parent node. The new node's weight is inserted in the queue, and the process continues until only a single weight remains in the queue. The last node is the root of the tree and the output of the algorithm. The root node is used as input to Algorithm 1 to generate the coding tree and grid

Algorithm 2 Huffman Tree

Input: $\mathcal{V}$; $\mathcal{P}$

```
1: //Generate tree nodes
2: for v_i ∈ V do
3:     Create a newNode(leftChild=None, rightChild = None,
4:                      parent = None, weight = p(v_i), code = ' ')
5: Insert nodes into priority queue Q
6: while len(Q)>1 do
7:     Sort Q in ascending order of weights
8:     (node_1, node_2) ← Extract first two nodes in Q
9:     Create a newNode(leftChild= node_1, rightChild = node_2,
10:                     parent = None,
11:                     weight = n_1.weight + n_2.weight, code = ' ')
12:    n_1.parent, n_2.parent= newNode
13:    Insert newNode into Q
14: //The last nodes in Q is the tree root
15: return root
```

indexes. The algorithm is executed with the time complexity of $O(n(\log_2 n))$.

The following steps illustrate the generation of Huffman tree for the example presented in Fig. 4.

1. One node is generated for each cell (v_1, v_2, v_3, v_4, v_5) and their probabilities are inserted in a priority queue

Q=($p(v_1)$=0.2, $p(v_2)$=0.1, $p(v_3)$=0.5, $p(v_4)$=0.4, $p(v_5)$=0.6).

2. The queue is sorted in an ascending order:

Q=($p(v_2)$=0.1, $p(v_1)$=0.2, $p(v_4)$=0.4, $p(v_3)$=0.5, $p(v_5)$=0.6)

3. The two nodes with the minimum weights (v_1 and v_2) are extracted from the queue and a new parent node r_1 is generated with the weight of $p(v_2) + p(v_1) = 0.3$ and inserted into the queue:

$$Q=(p(r_1)=0.3,\ p(v_4)=0.4,\ p(v_3)=0.5,\ p(v_5)=0.6)$$

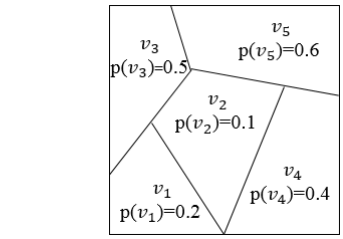
4. Similarly r_2, r_3, and r_4 are generated as

$$Q=(p(r_2)=0.7,\ p(v_3)=0.5,\ p(v_5)=0.6),$$
$$Q=(p(r_2)=0.7,\ p(r_3)=1.1),$$
$$Q=(p(r_4)=1.8).$$

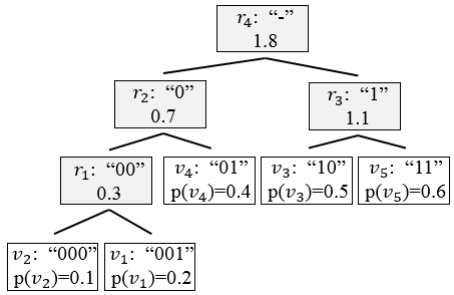
Another prefix tree evaluated in the experiments is called balanced tree. This prefix tree is used as a baseline to understand the improvement made by the Huffman tree. The balanced tree is a complete binary tree constructed in $log_2(n)$ steps. Given a tuple of probabilities corresponding to grid cells, they are sorted in ascending order and placed in a priority queue, i.e., Q. In the jth step, nodes $Q[2i]$ and $Q[2i+1]$ are paired for $i = 0, 1, ..., \lfloor n/2^j \rfloor - 1$, and each pair is replaced with a parent node in the queue. The weight of a parent is the addition of its immediate children's weights. The final remaining node in the queue is the tree's root.

3.3 Token Generation and Minimization

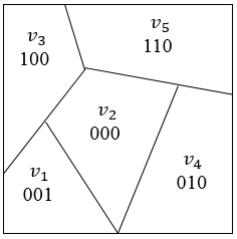
Prior work [14, 23] showed how the process of token generation for an alert zone can considerably improve the computation overhead, if the process of *token aggregation* is performed. Specifically, the binary codes corresponding to different regions of an alert zone can be aggregated to yield tokens with few non-star symbols, which in turn reduces the HVE overhead. Binary minimization on fixed-length codes is used for this purpose. For instance, suppose that the alert zone contains cells 0000, 0010, 0110, 0100,. Instead of separately encrypting the cell indexes and generating four



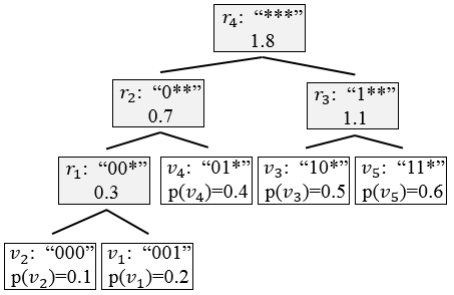
(a) Sample grid.



(b) Coding tree generated based on Huffman encoding.



(c) Assigned grid indexes.



(d) Coding tree.

Figure 4: Sample variable-length coding scheme

tokens, the TA uses binary minimization to generate a single token $0**0$, and the cost is reduced from twelve HVE operations to two. Binary minimization works when there are many cells in the alert zone, and when the placement of these cells permits code minimization. This approach is suitable when the number of alert cells is significant; however, in practice, alert zones may have cell configurations that do not permit efficient aggregation.

We propose a different token generation approach, where instead of performing binary minimization on fixed-length codes, we control the configuration of tokens based on the assignment of variable-length codes to cells. Algorithm 3 summarizes this process. Inputs to the algorithm are a set of alert cells and the coding tree. In the initialization phase, the algorithm defines:

- a dictionary of parent nodes (*parentDict*) with the number of leaf nodes in the corresponding subtree. This is done by traversing through children of parent nodes and counting the number of leaves located in that subtree. For the sample example, we have the dictionary as

$$[00* : 2,\ 0** : 3,\ 1** : 2,\ *** : 5]$$

- a list of leaf nodes denoted by *leaves*, ordered as they appear on the tree while traversing; no two edges of the tree cross path. Such a list for the sample tree is:

$$[v_2 : 000, v_1 : 001, v_4 : 01*, v_3 : 10*, v_5 : 11*].$$

The algorithm continues by converting alert cell indexes to codewords on the tree and recoding their associated codeword and the corresponding index in *leaves*. By default, the mapping process splits codewords into clusters that are located consecutively in *leaves*. It is important to note that mapping of alert cell indexes to codewords is unique, as demonstrated in Theorem 2. The theorem proves a bijective mapping between grid indexes and coding tree codewords. For instance, if the alert cells are $[001, 100, 110]$, then the mapping would result in leaves $[001, 10*, 11*]$ for the sample example. Next, the minimization process based on the coding scheme is conducted. The minimization's main idea is to find the common subtree roots that have maximum depths and use them as tokens. All leaves under a common subtree root must be alerted; otherwise, if a user is located in such a leaf node it will be falsely notified to be in an alert zone.

Continuing with the example and alert cells $[001, 10*, 11*]$, the algorithm generates two clusters $[10*, 11*]$ and $[001]$, and aims to identify the common subtree roots with the maximum depths in each cluster. This is done heuristically in lines 23- 37. Suppose that a cluster's length is L, the common left-hand side code in all L codewords is calculated and padded with '*' bits to ensure that the codeword length is RL. If the common codeword exists in the dictionary and the number of its children is L, the codeword is chosen as representative of its descendent leaves; otherwise, L is decremented by one, and now the first $L-1$ members are checked to see if there exists a common root associated with them. The process continues until the first subtree root is found. For the remaining codewords in the cluster, the algorithm is applied again until all tokens representing codewords in the cluster are selected. A similar approach is repeated for all clusters.

Algorithm 3 Deterministic Minimization

```
   Input: alertCells; codingTree;
1: parentDict = {}
2: for node ∈ codingTree do
3:     parentDict[node.code] = # descendent leaves
4: indexHolder, codewordHolder = []
5: leaves ← list of leaf codewords
6: for i ∈ alertCells do
7:     memCodeword ← Map i to a codeword in leaves
8:     codewordHolder = codewordHolder ∪{memCodeword}
9:     memIndex ← index of memCodeword in leaves
10:    indexHolder = indexHolder ∪{memIndex}
11: // Generate a two dimensional list of clusters
12: Clusters, c = []
13: c = c ∪ codewordHolder[0]
14: for i ∈ [1 : len(codewordHolder)] do
15:    if indexHolder[i] = indexHolder[i − 1] + 1 then
16:        c = c ∪ codewordHolder[i]
17:    else
18:        clusters = clusters ∪ c
19:        c = []
20:        c = c ∪ codewordHolder[i]
21: tokens = []
22: RL ← depth of tree
23: for cluster ∈ clusters do
24:    L = len(cluster)
25:    while L > 1 do
26:        code ← common bits in cluster[1 : L]
27:        if len(code) < RL then
28:            Pad with RL − len(code) star bits
29:        if code ∈ parentDict & parentDict[code] = L then
30:            tokens = tokens ∪ code
31:            cluster = cluster[L : len(cluster)]
32:            L = len(cluster)
33:        else
34:            L = L − 1
35:            if L = 1 then
36:                tokens = tokens ∪ cluster[L]
37:                cluster = cluster[L : len(cluster)]
38: return tokens
```

Theorem 2. *There exists a bijective function between grid indexes and the leaf nodes of the coding tree.*

Proof. We start by proving that for each index on the grid there exists a unique leaf node (codeword) on the tree. Let $\overline{x_1x_2...x_l}$ denote an arbitrary index on the map. There exists at least one leaf on the tree with the codeword $\overline{y_1y_2...y_{r_1} * ...*}$ such that $\overline{x_1x_2...x_{r_1}} = \overline{y_1y_2...y_{r_1}}$, as indexes have been generated from leaf nodes of the prefix tree. Suppose that there exist at least two leaf nodes with the codewords $\overline{y_1y_2...y_{r_1} * ...*}$ and $\overline{z_1z_2...z_{r2} * ...*}$ corresponding to the index $\overline{x_1x_2...x_l}$. Hence, we have the following relationship between the index and codewords on the tree.

$$\overline{x_1x_2...x_{r_1}} = \overline{y_1y_2...y_{r_1}} \tag{7}$$

$$\overline{x_1x_2...x_{r_2}} = \overline{z_1z_2...z_{r_2}} \tag{8}$$

Without loss of generality, assume that $r_2 \geq r_1$. Hence, equations 7 and 8 result in

$$\overline{y_1y_2...y_{r_1}} = \overline{z_1z_2...z_{r_1}}. \tag{9}$$

However, this contradicts the prefix property of the codes. Hence, there is a unique leaf node corresponding to each cell index. As there are an equal number of indexes and codewords, there exists a bijective mapping between indexes and codewords. □

4 EXTENSION TO NON-BINARY CODES

So far, we considered the alphabet of HVE operations to be limited to $\Sigma = \{0, 1\}$ and the extended alphabet as $\Sigma_* = \Sigma \cup \{*\}$. This is an intuitive way of looking at indexes as they are a series of zeros and ones. However, by extending the alphabet to $\Sigma = \{0, 1, ..., B-1\}$ for an arbitrary integer $B \in \{2, ..., n-1\}$, we could obtain more compact representations. The special character is also added as $\Sigma_* = \Sigma \cup \{*\}$. We re-visit the operations from the previous section for the extended alphabet with B symbols.

1. *Prefix tree:* We incorporate an extension of Huffman coding referred to as B-ary Huffman to generate the prefix tree. The main idea is to group B least probable symbols (instead of 2) at each substitution stage of the algorithm. The construction of the prefix tree for our running example grid is shown in Fig. 6a in which a 3-ary or Huffman code is used. Initially, the algorithm

(a) (b)

Figure 5: Expansion process.

starts by combining nodes v_2, v_1, and v_4, as they correspond to a group of three nodes with the minimum total weight, generating the node r_1. Next, the nodes r_1, v_3, and v_5 are combined, and the root node r_2 is generated. The weights and other characteristics of the nodes are stored and calculated in the same way as the binary Huffman tree. The codes associated with the tree are generated by assigning an empty character to the root node and then traversing the tree. At each branching node, when following the ith child edge, character $i-1$ is added to the root string. As an example, prefix code '02' is generated by adding character '0' at r_1, and character '2' by moving to node v_4. As in the case of the binary case, we are interested in codes assigned to internal nodes as well as the prefix codes generated at the leaves.

2. *Coding tree:* The generation of the coding tree requires an additional step compared to the binary Huffman tree. In the first step, codes are padded with star characters until they reach the same length as the RL. The padded prefix tree for our running example is shown in Fig. 6b. Next, we expand each character to an array of B bits. The character $i \in \Sigma$ is converted to B bits with the $(i+1)$-th bit set to 1 and star bits otherwise. The only exception is the star character, which will be mapped to a string of length B with all bits set to '*'. As an example, the expansion of $2*$ is shown in Fig. 5a.

Each original character essentially works as a placeholder for the expanded representation. The final coding tree generated for our example is shown in Fig. 6c.

3. *Indexes:* We generate indexes by padding the leaves in the prefix tree by zeros, and then expanding the codes. An interesting case occurs that gives the TA the opportunity to increase the grid's granularity further if desired. Consider the prefix code '2', which will be zero-padded to generate '20'. The expansion process requires two steps: (i) zeros generated by the padding process are mapped to B bits; (ii) each character $i \in \Sigma$ is expanded to B bits with the $(i+1)$-th bit set to 1, and star bits otherwise. The expansion of '20' is demonstrated in Fig. 5b.

The additional star bits in the index are converted to zeros. The advantage of the approach is revealed when we increase the granularity of a grid cell in a later stage in time. This can be done by exploiting the star bits generated in the last step without violating the structure of the grid or the coding tree. Consider the index '20' corresponding to cell v_5 one more time. This string was first converted to '**1000' and then to '001000'. Suppose, later on, the TA decides to increase the granularity of v_5 to four cells. This can simply be done by using four indexes '001000', '011000', '101000', '111000' generated based on star bits with all of them lying under character '2'. The coding tree is also updated accordingly via placeholders for the character '2' without violating the tree's prefix property.

5 ENCRYPTION OVERHEAD

Employing variable-length codes into HVE can significantly improve the computation complexity at the SP, but there is a a trade-off with respect to increased encryption time. When variable-length encoding is used, all ciphertexts submitted by the mobile users to the SP must have the maximum length of any existing code. Otherwise, the length of the ciphertext would enable the SP to pinpoint the location of the submitting user to one of the cells that are assigned a code with bit length equal to the one submitted. To thwart such attacks, all codes are padded before encryption to the maximum possible length, i.e. RL. In this section, we analyze this additional encryption overhead, and we show that it is not significant, especially compared to the savings at the SP. Furthermore, the additional computational load is spread over the user population, since each user encrypts its own location independently, and no bottleneck is created (as opposed to the alert matching overhead which is centrally incurred at the SP).

In our analysis, we make use of the following result:

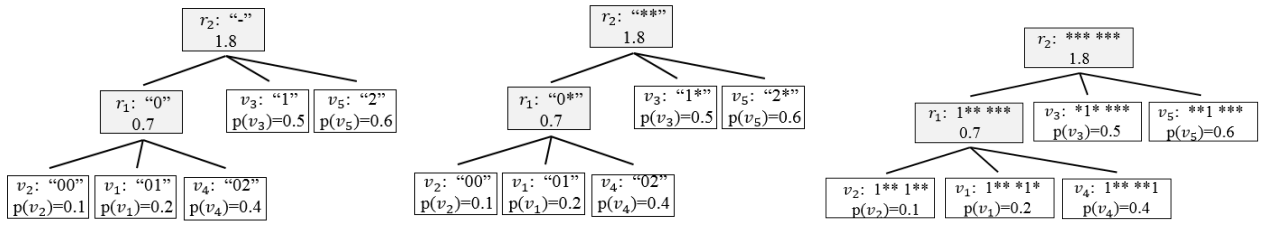
THEOREM 3. *The depth of a B-ary Huffman tree (RL) with n leaves is less than or equal to* $\lceil \frac{n-1}{B-1} \rceil$.

PROOF. The theorem can be proved by counting the number of internal nodes in a B-ary Huffman tree. Consider a tree with n leaves generated by the Huffman mechanism. At every run of the algorithm, B less likely remaining nodes in the priority queue are combined, and a new internal node is inserted. Suppose that the Huffman mechanism is conducted x times over the priority queue until a single node, i.e. root node, is left in the queue. The maximum value of integer x can be derived as:

$$\max_x \ \{n - x(B-1) \geq 1\} \rightarrow x = \lceil \frac{n-1}{B-1} \rceil \qquad (10)$$

Therefore, the maximum possible depth of a B-ary Huffman tree is $\lceil \frac{n-1}{B-1} \rceil$. □

Let L_E denote the difference between the RL of an encoding grid with n cells generated by Huffman coding and fixed-length codes. We start by deriving an upper bound for L_E when $\Sigma_* = \{0, 1\} \cup \{*\}$, and then extend the upper bound for an arbitrary size alphabet. Without loss of generality, consider that RL in the binary Huffman tree is l_n. The minimum possible value for l_n is



(a) Ternary Huffman coding tree. (b) Placeholders. (c) Coding tree.

Figure 6: Sample coding tree for extended framework.

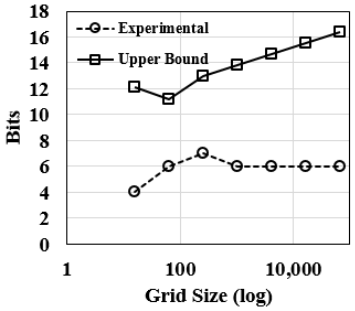


Figure 7: Upper bound of L_E for Binary Huffman codes.

$\lceil \log_2 n \rceil$. Based on Theorem 3, L_E can be written as:

$$L_E(B=2,n) = l_n - \lceil \log_2 n \rceil \leq \lceil \frac{n-1}{2-1} \rceil - \lceil \log_2 n \rceil = n-1-\lceil \log_2 n \rceil \tag{11}$$

A tighter upper-bound for RL in a binary Huffman tree can be derived based on the following theorem proven in [6] (we omit the proof for brevity):

Theorem 4. *Let p_n and l_n denote the minimum probability and its corresponding length existing on the Huffman tree. Then,*

$$l_n \leq log_\phi \frac{1}{p_n} \tag{12}$$

where ϕ denotes the golden ratio*, i.e., $\phi = (1+\sqrt{5})/2$.*

Therefore, a tighter upper-bound for L_E can be written as

$$L_E(B=2,n) \leq log_\phi \frac{1}{p_n} - \lceil \log_2 n \rceil \tag{13}$$

The numerical and analytical values of L_E are verified[1] for binary Huffman coding in Fig. 7.

Now let us extend the approach for B-ary Huffman codes generated with the alphabet $\Sigma_* = \{0, 1, ..., B-1\} \cup \{*\}$. Based on information theory, the minimum length of RL corresponding to fixed-length codes is derived as $\lceil \log_B n \rceil$. Therefore, the upper-bound for L_E can be computed as:

$$L_E(B,n) = B(l_n - \lceil \log_B n \rceil) \leq B(\lceil \frac{n-1}{B-1} \rceil - \lceil \log_B n \rceil) \tag{14}$$

$$\leq B(\frac{n-1}{B-1} + 1 - \lceil \log_B n \rceil) \tag{15}$$

The multiplier B is required to map the alphabet to 0s and 1s, used in the encryption.

$$E[L_E(n)] \leq \frac{1}{n-1}(\sum_{i=2}^{n} \frac{i(n-1)}{i-1} + \sum_{i=2}^{n} i - \sum_{i=2}^{n} i\lceil \log_i n \rceil) \tag{16}$$

The first and second summation in the upper-bound of $E[L_E(n)]$ can be further simplified as

$$\sum_{i=2}^{n} \frac{i(n-1)}{i-1} = (n-1) \times (n-1+\sum_{i=2}^{n} \frac{1}{i-1}) \tag{17}$$

$$\approx (n-1) \times (n-2+\ln(n-1) + \frac{1}{2(n-1)} + \gamma) \tag{18}$$

and,

$$\sum_{i=2}^{n} i = \frac{n^2+n-1}{2} \tag{19}$$

[1]Grid probabilities are generated with the parameters of sigmoid function set to $a = 0.95$ and $b = 20$. Please refer to Section 6 for details.

where $\gamma \approx 0.577$ is the *Euler-Mascheroni* constant. The approximation for the nth Harmonic can be derived by its asymptotic expansion in the *Hurwitz zeta* function [8].

6 SECURITY DISCUSSION

Our proposed technique uses as building block HVE primitives as introduced in [5], and hence inherits the security properties of HVE, namely IND-CCA under the bilinear Diffie-Hellman assumption. In terms of ciphertext processing semantics, the security achieved by our technique is similar to existing work in the area of secure computation, namely the only leakage that occurs as part of ciphertext matching is the evaluation outcome. Specifically, the SP learns only whether the user is included in the alert zone (which is a necessary condition for correctness), and no other information. The SP does not learn where exactly the user is located within the alert zone, if the match is successful; conversely, if the match is not successful, the SP learns only that the user is not inside the alert zone, but cannot further narrow down the user within the data domain.

Furthermore, our technique is guided by statistical information that is derived solely from public data. Namely, the heuristic on how to encode cells does not use any user location data, but strictly likelihood scores that are assigned to grid cells, based on public knowledge regarding the *alert zone* properties, such as site popularity, etc. No private information regarding any system user is included in the encoding process (not even aggregate data, such as user distribution, etc).

Finally, the encryption strength achieved by HVE depends on the underlying bilinear pairing curve used [5]. Modern elliptic-curve pairing-based cryptography can easily provide 128-bit security, which is on par with commercial database applications such as banking, or heathcare data security standards.

7 EXPERIMENTAL EVALUATION

We conduct our experiments on a 3.40GHz core-i7 Intel processor with 8GB RAM running 64-bit Windows 7 OS. The code is implemented in Python. We evaluate our methods on both real and synthetic datasets, as follows:

- *Chicago Crime Dataset.* This dataset is provided by the Chicago Police Department's CLEAR (Citizen Law Enforcement Analysis and Reporting) system [1]. The dataset consists of reported incidents of crime that occurred in the city of Chicago in 2015. We consider four categories of crime: homicide, sexual assault, sex offense, and kidnapping. Fig. 8 shows data statistics. A 32×32 grid is overlaid on top of the dataset, and a logistic regression model is trained with the crime data from January to November 2015, and tested on the December data. The accuracy of the model is 92.9% and the generated likelihood scores based on the model are used as input to our techniques.
- *Synthetic data.* We generate the likelihood of grid cells to be part of an alert zone using a sigmoid activation function $\mathcal{S}(X=x) = 1/(1+\exp^{-b(x-a)})$, where a and b are parameters controlling the function shape. For each data point (i.e., cell) x, a uniformly random number between zero and one is generated, i.e., $x \in X \sim$ uniform$(0,1)$. Then, the number is fed into the sigmoid activation function. The output is a value between zero and one indicating the likelihood of the cell to be inside an alert zone. The sigmoid function is a frequent model used in machine learning, and

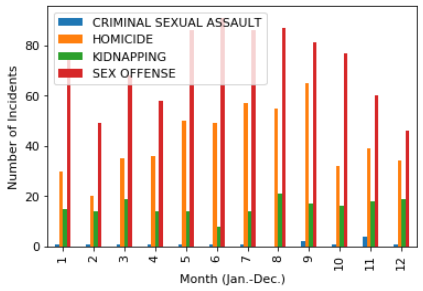
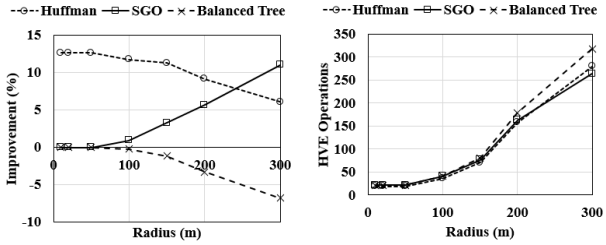


Figure 8: Chicago crime dataset statistics.



(a) (b)

Figure 9: Evaluation on Chicago crime dataset.

we choose it because we expect that, in practice, the probability of individual cells becoming part of an alert zone can be computed using such a model built on a regions' map of features (e.g., type of terrain, building designation, etc.). Parameter a of the sigmoid controls the *inflection* point of the curve, whereas b controls the gradient.

We compare our proposed variable-length encoding scheme with the state-of-the-art fixed-length approach scaled gray optimizer (SGO) from [23], which uses graph embedding to reflect cell probabilities in the way cell codes are chosen. We also consider as a second benchmark an approach that uses balanced trees, as opposed to Huffman trees.

We use as performance metric the number of HVE bilinear map pairing operations incurred by each technique (which are the most expensive component of the overhead). We present both absolute counts, as well the percentage of *improvement* compared to the original fixed-length encoding HVE approach introduced in [14] (which assumes all cells are equally likely to be alerted).

7.1 Evaluation on Real Dataset

Fig. 9 shows the performance results obtained on the real dataset. The x-axis in each graph indicates the size of the alert zone (expressed as radius). For low radii values, the SGO algorithm fails to provide significant improvement, due to the fact that the binary minimization process used by fixed-length encoding approaches is unable to aggregate tokens. In contrast, the proposed variable-length technique using Huffman encoding is able to provide gains of up to 15% compared to the baseline. In practice, we expect alert zones to be relatively compact compared to the data domain, hence this case is frequently occurring in practice. Furthermore, the results show the superiority of the Huffman code compared to generic variable-length encodings, as the balanced-tree approach benchmark does not produce any improvement.

As the size of alert zone increases, SGO improves, whereas the gain of Huffman encoding decreases. This is expected, since with very large alert zones, it is easy to aggregate tokens, by grouping together cells with low Hamming distance between their codes. However, such an improvement can only be reached when the alert zones are very large, which is not a realistic scenario in practice. In general, the size of alert zones is expected to be small, and their distribution in the data domain sparse, which would further diminish the potential of SGO (and other binary minimization approaches) to produce performance gains, as aggregation requires clustered cells with similar binary codes.

7.2 Evaluation on Synthetic Dataset

Performance evaluation results for synthetic data are summarized in Fig. 10. We use two inflection points for the sigmoid function $a = 0.90, 0.99$, as well as three gradient values $b = 10$, $b = 100$ and $b = 200$. A similar trend to the real dataset is observed. The Huffman tree approach achieves significantly better performance when the alert zones are compact, which is the expected case in practice.

Two other trends can be observed with respect to the parameters of the sigmoid function. First, a higher inflection point setting results in a more skewed distribution probability on the grid, and leads to a higher performance gain for Huffman encoding compared to competitor approaches. The performance gain can be as high as 50%. This is a positive aspect, since in real life one expects alert cell probabilities to be quite skewed, where more popular areas are visited by more individuals, hence there is more potential for alert events (e.g., public-safety alerts, or visits of a COVID-infected patient to points of interest). Second, an increase in the gradient of initial probabilities (b) also improves the performance gain of Huffman encoding.

We also conducted an experiment under mixed-workload conditions. We consider several mixes between short-radius (20 meters) and long-radius (300 meters) alert zones: *W1* (90% short-10% long); *W2* (75% short-25% long); *W3* (25% short-75% long); and *W4* (10% short-90% long). Results are summarized in Fig. 11. Our proposed technique outperforms SGO for all considered cases. For mostly-compact alert zones (*W1*), the improvement is much higher than that of SGO, with absolute values of up to 40%.

On the synthetic data, we are also able to perform more in-depth tests where we vary the parameter settings of our proposed approach. In Fig. 12, we vary the grid granularity. The results are obtained for $a = 0.95$ and $b = 20$. The results show that higher grid granularities lead to higher performance overhead, which is expected, since more cells need to be encoded and encrypted, and thus code lengths increase. We also observe an interesting trend: the improvement for a low number of alert cells decreases at higher granularity levels. As the number of grid cells grow, and considering the same sigmoid activation function parameters, there will be more cells with low probabilities of becoming an alert cell. Therefore, the Huffman tree tends to have higher depths. This can be observed more accurately in Fig. 13, where we show the ratio of average length to the maximum length of the Huffman tree for various grid sizes. Hence, the improvement achieved by deterministic minimization lags behind the logic minimization approach, leading to a smaller improvement percentage.

Finally, we present the run time required to generate indexes and the coding tree in Fig. 14. Note that, this is a one-time setup cost, as the process is only run when initializing the system, and has no effect on run-time performance. In the worst case, the process takes minutes for larger-granularity grids.

8 RELATED WORK

Location Privacy. Early works on location data privacy pivoted around the k-anonymity [25] model. The main idea is to hide users' location among at least k-1 other users to protect user privacy. A preliminary approach to achieve k-anonymity

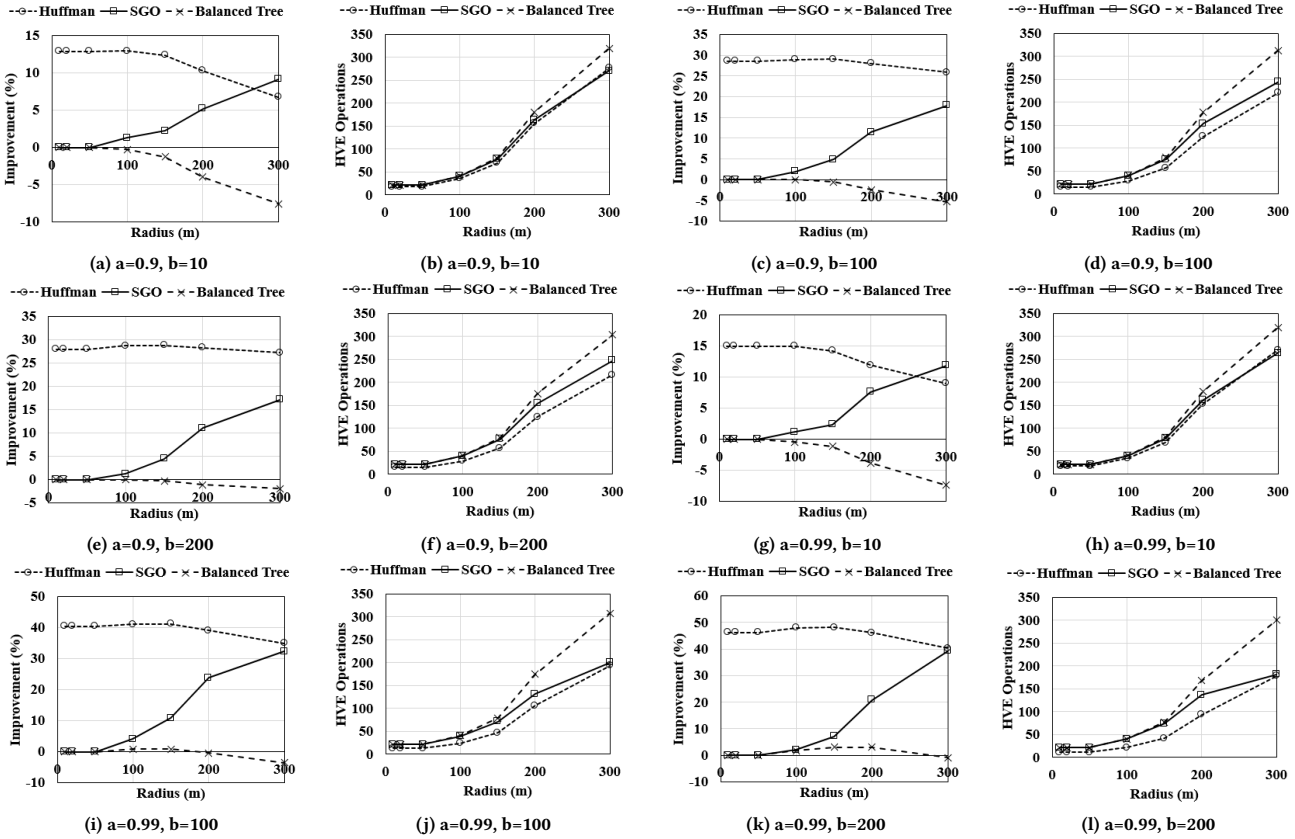


Figure 10: Performance evaluation on synthetic dataset.

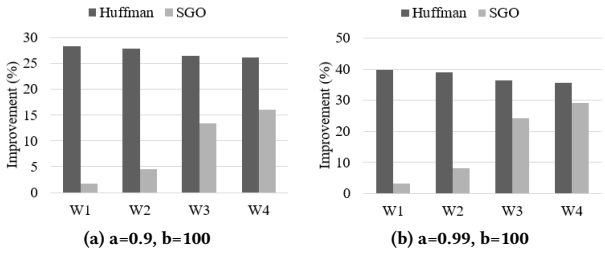


Figure 11: Mixed workloads, synthetic dataset.

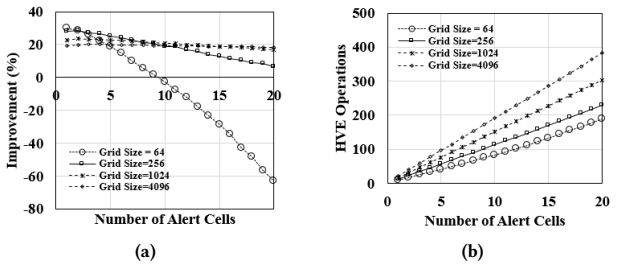


Figure 12: Varying grid granularity, synthetic dataset.

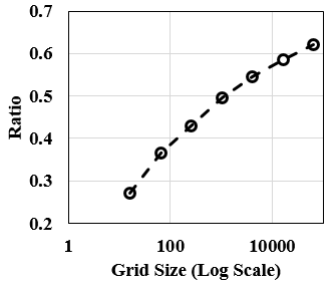


Figure 13: Average-to-maximum code length ratio.

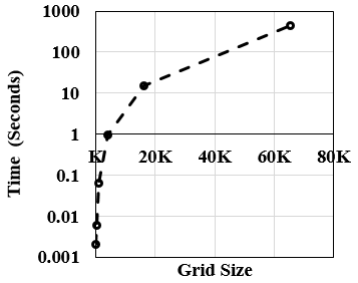


Figure 14: System Initialization Time

was focused on the generation dummy (fake) locations for data points [18]. Unfortunately, dummy generation algorithms are shown to be susceptible to inference attacks [22].

An alternative proposed method to achieve k-anonymity has been focused on the concept of *Cloaking Regions (CRs)* [15]. Most approaches in this category take advantage of a trusted anonymizer to generate a cluster of k user locations and query the area locations are enclosed by, achieving k-anonymity [13, 17, 20]. Approaches based on CRs are effective in a single snapshot [17]; however, once users are considered in trajectories, requiring continuous queries, privacy concerns are posed on the system by inference attacks. Moreover, large CRs are needed in trajectories, significantly reducing the utility of data [9] as well as posing privacy risks due to inference attacks. The authors in [12, 16]

aim at providing privacy by distinguishing between sensitive and non-sensitive locations based on user preferences.

Searchable Encryption The main motivation behind searchable encryption techniques is outsourcing the data management to a third party, such as cloud providers without the third party learning about data or queried information by users. The use of a searchable encryption was initially proposed in [24] for a secure cryptographic search of keywords. The approach supports comparison queries [4] as well as subset queries and conjunctions of equality [5]. The concept of HVE used in this paper was first proposed in [5] and later extended in [3]. The authors in [14] proposed the use of HVE to guarantee user privacy in location-based alert systems. Despite promising results of the approach, a major challenge is reducing the computation complexity of HVE at the server where the matching process is conducted. The work in [23] represents the current state-of-the art in location-based alerts with searchable encryption, and it takes into account probabilities of cells being part of an alert zone. A graph embedding technique is used to assign codes to cells in a manner that is aware of their likelihood of becoming alerted. The approach achieves significant improvement in performance compared to [14]. However, as our experimental evaluation shows, such improvements are reached only when a relatively large number of alert cells are part of an alert zone. For alert zones with few cells, our approach clearly outperforms that of [23].

9 CONCLUSIONS AND FUTURE WORK

We proposed a technique for secure location-based alerts that uses searchable encryption in conjunction with variable-length location encoding. Specifically, using Huffman compression codes, we showed that it is possible to significantly reduce the overhead of searchable encryption for cases where alert zones are compact and sparse, which is the case we believe to be most likely in practice. Extensive analytical and empirical evaluation results prove that our proposed approach significantly outperforms existing fixed-length encoding techniques, with only a small overhead in terms of additional encryption time.

In some cases, our approach may be limited by the lack of a systematic way of obtaining the probability values for various data domain regions. While having accurate probabilities is a plus, we do not require high accuracy in the actual values. In fact, in our design it is often the relative ordering of the probabilities that matters, and not necessarily the exact values. In practice, one can produce a relatively stable and representative ordering of types of features based on their popularity. Even without precise probability values, one can still obtain significant gains.

In future work, we plan to investigate more advanced stochastic models that capture correlations between cells in an alert zone, as well as cases when the alert zone evolution over time can be estimated by a spread model (e.g., a chemical gas leak). Significant performance gains can be achieved in such scenarios. One possibility is to model the space and time based on a Markov model. For a grid with n cells, the model would consist of 2^n states, each representing a unique subset of grid cells. Next, one can determine a stationary distribution of probabilities over cells, and derive the values required to reach equilibrium.

Finally, while our work focuses on location data, our design can be extended to benefit other types of data as well. Our assumed semantics for ciphertext processing is that of range queries, and numerous other data types can benefit from secure range queries. However, one has to devise specific encodings and optimizations for each type of data, as straightforward application of HVE to generic data types may lead to high performance overhead, as illustrated in our earlier work [14].

Acknowledgment. This research has been funded in part by NSF grants IIS-1910950, IIS-1909806 and CNS-2027794, the USC Integrated Media Systems Center (IMSC), and unrestricted cash gifts from Google and Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the sponsors such as the NSF.

REFERENCES

[1] https://data.cityofchicago.org/public-safety/crimes-2015/vwwp-7yr9.
[2] T. Almanie, R. Mirza, and E. Lor. Crime prediction based on crime types and using spatial and temporal criminal hotspots. *arXiv preprint 1508.02050*, 2015.
[3] C. Blundo, V. Iovino, and G. Persiano. Private-key hidden vector encryption with key confidentiality. In *International Conference on Cryptology and Network Security*, pages 259–277. Springer, 2009.
[4] D. Boneh, A. Sahai, and B. Waters. Fully collusion resistant traitor tracing with short ciphertexts and private keys. In *Intl. Conf. on the Theory and Applications of Cryptographic Techniques*, pages 573–592. Springer, 2006.
[5] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *Theory of Cryptography Conference*, pages 535–554. Springer, 2007.
[6] M. Buro. On the maximum length of huffman codes. *Information processing letters*, 45(5):219–223, 1993.
[7] R. Chandrasekharam, V. Vinod, and S. Subramanian. Genetic algorithm for embedding a complete graph in a hypercube with a vlsi application. *Microprocessing and microprogramming*, 40(8):537–552, 1994.
[8] C.-P. Chen and J.-X. Cheng. Ramanujan's asymptotic expansion for the harmonic numbers. *The Ramanujan Journal*, 38(1):123–128, 2015.
[9] C.-Y. Chow and M. F. Mokbel. Enabling private continuous queries for revealed user locations. In *Symp. on Spatial and Temporal Databases*, page 258, 2007.
[10] T. M. Cover. *Elements of information theory*. John Wiley & Sons, 1999.
[11] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.
[12] M. Damiani, E. Bertino, and C. Silvestri. Probe: an obfuscation system for the protection of sensitive location information in lbs. *TR2001-145, CERIAS*, 2008.
[13] B. Gedik and L. Liu. Location privacy in mobile systems: A personalized anonymization model. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 620–629. IEEE, 2005.
[14] G. Ghinita and R. Rughinis. An efficient privacy-preserving system for monitoring mobile users: making searchable encryption practical. In *Proc. of ACM Conf. on Data and application security and privacy*, pages 321–332. ACM, 2014.
[15] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 31–42, 2003.
[16] M. Gruteser and X. Liu. Protecting privacy, in continuous location-tracking applications. *IEEE Security & Privacy*, 2(2):28–34, 2004.
[17] P. Kalnis, G. Ghinita, K. Mouratidis, and D. Papadias. Preventing location-based identity inference in anonymous spatial queries. *IEEE transactions on knowledge and data engineering*, 19(12):1719–1733, 2007.
[18] H. Kido, Y. Yanagisawa, and T. Satoh. An anonymous communication technique using dummies for location-based services. In *ICPS'05. Proceedings. International Conference on Pervasive Services, 2005.*, pages 88–97. IEEE, 2005.
[19] S. Lai, S. Patranabis, A. Sakzad, J. K. Liu, D. Mukhopadhyay, R. Steinfeld, S.-F. Sun, D. Liu, and C. Zuo. Result pattern hiding searchable encryption for conjunctive queries. In *Proc. of ACM CCS*, pages 745–762, 2018.
[20] M. F. Mokbel, C.-Y. Chow, and W. G. Aref. The new casper: Query processing for location services without compromising privacy. In *Proceedings of the 32nd international conference on Very large data bases*, pages 763–774, 2006.
[21] K. Nguyen, G. Ghinita, M. Naveed, and C. Shahabi. A privacy-preserving, accountable and spam-resilient geo-marketplace. In *Proc. of ACM SIGSPATIAL*, pages 299–308. ACM, 2019.
[22] S. Shaham, M. Ding, B. Liu, S. Dang, Z. Lin, and J. Li. Privacy preservation in location-based services: a novel metric and attack model. *IEEE Transactions on Mobile Computing*, 2020.
[23] S. Shaham, G. Ghinita, and C. Shahabi. Enhancing the performance of spatial queries on encrypted data through graph embedding. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 289–309. Springer, 2020.
[24] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.
[25] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

Concealer: SGX-based Secure, Volume Hiding, and Verifiable Processing of Spatial Time-Series Datasets

Peeyush Gupta, Sharad Mehrotra, Shantanu Sharma, Nalini Venkatasubramanian, and Guoxi Wang
University of California, Irvine, USA.
sharad@ics.uci.edu, shantanu.sharma@uci.edu

ABSTRACT

This paper proposes a system, entitled Concealer that allows sharing time-varying spatial data (*e.g.*, as produced by sensors) in encrypted form to an untrusted third-party service provider to provide location-based applications (involving aggregation queries over selected regions over time windows) to users. Concealer exploits carefully selected encryption techniques to use indexes supported by database systems and combines ways to add fake tuples in order to realize an efficient system that protects against leakage based on output-size. Thus, the design of Concealer overcomes two limitations of existing symmetric searchable encryption (SSE) techniques: (*i*) it avoids the need of specialized data structures that limit usability/practicality of SSE in large scale deployments, and (*ii*) it avoids information leakages based on the output-size, which may leak data distributions. Experimental results validate the efficiency of the proposed algorithms over a spatial time-series dataset (collected from a smart space) and TPC-H datasets, each of 136 Million rows, the size of which prior approaches have not scaled to.

1 INTRODUCTION

We consider the problem wherein trusted data producers ($\mathcal{DP}$) share users' spatial time-series data in the encrypted form with untrusted service providers ($\mathcal{SP}$) to empower $\mathcal{SP}$ to build value-added applications for users. Examples include a cellular company sharing data about the cell tower a user's mobile phone is connected to, or an organization/university providing WiFi connectivity data about the access point a user's device is connected to, for applications such as building dynamic occupancy maps [1]. We classify applications supported by $\mathcal{SP}$ using user's data into two classes, as follows:

(1) **Aggregate Applications** that aggregate data of multiple users to build novel applications. Examples include occupancy of different regions, heat maps, and count of distinct visitors to a given region over a period of time. Such applications are already supported by several service providers, *e.g.*, Google Maps supports information about busy-status and wait times at stores such as restaurants and shopping malls.

(2) **Individualized Applications** that allow users to ask queries based on their own past movements, *e.g.*, locations a person spent the most time during a given time interval, finding the number of people came in contact with, and/or other aggregate operations on user's data. Such applications can be useful for several contexts including exposure tracing in the context of infectious diseases [37].

Implementing applications at the (untrusted) $\mathcal{SP}$ requires: (*i*) $\mathcal{DP}$ to appropriately encrypt data prior to sharing with $\mathcal{SP}$, (*ii*) $\mathcal{SP}$ to be able to execute queries on behalf of the user over the encrypted data, and (*iii*) the user to be able to decrypt the encrypted answers returned by *SP*. Realizing such a data-sharing architecture leads to the following three requirements, (of which the first two are relatively straightforward, while the third requires a careful design of a new cryptographic technique that this paper focuses on):

R1: Query formulation by the user. Given that data is encrypted by $\mathcal{DP}$ and is hosted at $\mathcal{SP}$, the user needs to formulate the query to enable $\mathcal{SP}$ to execute it over encrypted data. The users can formulate an appropriate encrypted query, if they know the key used for encryption by $\mathcal{DP}$. However, $\mathcal{DP}$ cannot share the key with users to prevent them from decrypting the entire dataset. A trivial way to overcome this problem is to involve $\mathcal{DP}$ in processing queries. Particularly, a user can submit queries to $\mathcal{DP}$ that converts the query into appropriate trapdoors to be executed on encrypted data at $\mathcal{SP}$, fetches the partial results from $\mathcal{SP}$, and processes the fetched rows, before producing the final answer to users. Such an architecture incurs significant overhead at $\mathcal{DP}$ and requires $\mathcal{DP}$ to mediate each user query, (pushing $\mathcal{DP}$ to act as a surrogate $\mathcal{SP}$). Thus, ***the first requirement is how users can formulate appropriate encrypted queries without involving $\mathcal{DP}$ in query processing.***

We can overcome this requirement *trivially* by using secure hardware (*e.g.*, Intel Software Guard eXtensions[1] (SGX) [9]) at $\mathcal{SP}$ that works as a trusted agent of $\mathcal{DP}$. SGX receives queries encrypted using the public key of SGX (which we assume to known to all) from users, decrypts the query, converts the query into appropriate secure trapdoors, and provides the answer.

R2: Preventing $\mathcal{SP}$ from impersonating a user. Since we do not wish to involve $\mathcal{DP}$ during query processing, all users ask queries directly to $\mathcal{SP}$. Such query representations should not empower $\mathcal{SP}$ to mimic/masquerade as a legitimate user to gain access to the cleartext data from the answers to the query. Thus, ***the second requirement is how will the system prevent $\mathcal{SP}$ to mimic as a user and execute a query.***

We overcome this requirement *trivially* by building a list of *registered users*, who are allowed to execute queries on the encrypted data (after a negotiation between users and $\mathcal{DP}$) at $\mathcal{DP}$ and provides it in encrypted form to $\mathcal{SP}$. The registry contains credential information (*e.g.*, public/private key and authentication information of users) about the users who are interested

This material is based on research sponsored by DARPA under Agreement No. FA8750-16-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work is supported by NSF Grants 1527536, 1545071, 2032525, 1952247,1528995, 2008993.

[1] Recent Intel CPUs introduced SGX that allows us to create a small trusted execution environment, called *enclave* that is isolated and protected from the rest of the system. SGX protects computations from the operating system (controlled by the third-party) and from numerous applications/system-level attacks. Unfortunately, existing implementations of SGX are prone to side-channel attacks that exploit one of the microarchitectural components of CPUs, *e.g.*, cache-lines, branch execution, page-table access [20, 39, 40], and power attacks. Nevertheless, systems T-SGX [34] and Sanctum [10] have evolved to overcome such attacks, and it is believed that future versions of SGX will be resilient to those attacks.

Techniques	Frequent and fast insertion	Fast query execution	DBMS supported index	Preventing attacks		
				Data distribution	Output-size	Access-patterns
DET (Always Encrypt [3])	1	1	Yes	No	No	No
NDET (Arx [32] or Always Encrypt [3])	2	2 or 3	Yes	Yes	No	No
Indexable-SSE (PB [22]- or IB [23]-Tree)	4	2	No*	No	No	No
Indexable-SSE with ORAM (Blind Seer [30])	4	4	No*	No	No	Yes
Non-indexable-SSE [11, 35]	2	4	No	No	No	No
SGX system (Opaque [42])	2	3	No	No	No	No
MPC or SS (Jana [4])	4	4	No	Yes	No	Yes
Concealer	1	1	Yes	Yes	Yes	Yes (partial)

Table 1: Comparing different techniques vs Concealer. Note: 1: Very fast, 2: Fast, 3: Slow. 4: Very slow. ∗: Indexable SSEs build their own indexes and their index traversal techniques are not in-built in existing commercial DBMS.

in $\mathcal{SP}$ applications. Thus, before generating any trapdoor by SGX, it first authenticates the user and provides the final answers encrypted using the public key of the user.

R3: Selecting the appropriate encryption technique. Spatial time-series data brings in new challenges (as compared to other datasets) in terms of a large-amount of the dataset and dynamically arriving data. Also, spatial time-series data show new opportunities in terms of limited types of queries (*i.e.*, not involving complex operations such as join and nested queries). Particularly, the data encryption and storage must sustain the data generation rate, *i.e.*, the encryption mechanism must support dynamic insertion without the high overhead. Further, cryptographic query execution time should scale to millions of records. Finally, the system must support strong security properties such that the ciphertext representation and query execution do not reveal information about the data to $\mathcal{SP}$. Note that ciphertext representation leaks data distribution only when deterministic encryption (DET) is used. Query execution leaks information about data due to search- and access-patterns leakages, and volume/output-size leakage. In §1.1, we discuss these leakages and argue that none of the existing cryptographic query processing techniques satisfy all the above requirements. Thus, ***the third requirement is how to design a system that has efficient data encryption and query execution techniques, and not prone to such leakages.***

Concealer. We design, develop, and implement a secure spatial time-series database, entitled Concealer. This paper focuses on how Concealer addresses the above-mentioned third requirement, and below, we briefly discuss the proposed solution to the requirement R3. Concealer is carefully designed to support a high rate of data arrival, and large data sets, but it only supports a limited nature of spatial time-series queries required by the domain of interest. To a degree, Concealer can be considered more of a vertical technology compared to general-purpose horizontal solutions, which as will be discussed in §1.1, lack the ability to support application/data that motivates Concealer.

Concealer, for fast data encryption and minimum cryptographic overheads on each tuple, uses a variant of deterministic encryption that produces secure ciphertext (that does not reveal data distribution) and is fast enough to encrypt tuples (≈37,185 tuples/min). Further, Concealer exploits the index supported by MySQL. Note that we do not use any specialized index (*e.g.*, PB-tree [22] and IB-Tree [23]) and do not require to build the entire index for each insertion at the trusted side. Since Concealer users an index supported by DBMS, it supports efficient query execution. For a point query on 136M rows, Concealer needs at most 0.9s. Thus, our implementation of DET and the use of indexes supported by DBMS satisfy the requirements of fast data insertion and fast query execution.

To address the security challenge during query execution, Concealer (*i*) prevents output-size by fixing the unit of data retrieval of the form of bins, formed over the tuples of a given time period; care is taken to ensure that each bin must be of identical size (by implementing a variant of bin-packing algorithm [8]), and (*ii*) hides *partial* access-patterns, due to retrieving a fixed bin having different tuples corresponding to different sensor readings (with different location/time/other values) for any query corresponding to the element of the bin. That means the adversary observes: which fixed tuples are fetched for a set of queries including the real query posed by the user. However, the adversary cannot find which of the fetched tuples satisfy the user query. Since our focus is on practical system implementation, we relax the complete access-pattern hiding requirements. The exact security offered by Concealer will be discussed in §7.

Since we fetch a bin of several tuples, to filter the useless tuples that do not meet the query predicates, SGX at $\mathcal{SP}$ filters them, (while also hides complete access-patterns inside SGX by performing oblivious operations). Further, to verify the integrity of the data before producing the answer, Concealer provides a *non-mandatory* hash-chain-based verification.

Evaluation. We evaluate Concealer on a real WiFi dataset collected from at UCI. To evaluate its scalability, we executed the algorithms on 136M rows, the size that previous existing cryptographic techniques cannot support. We also compare Concealer against SGX-based Opaque [42]. To the best of our knowledge, there is no system that supports identical security properties (hiding output-size and hiding partial access-patterns, while supporting indexes for efficient processing). Our algorithms can be used to deal with non-time-series datasets also; thus, to evaluate algorithms' practicality, we evaluate aggregation queries on 136M rows LineItem table of TPC-H benchmark.

1.1 Comparison & Advantages of Concealer

We discuss common leakages from cryptographic solutions, argue that they do not satisfy the requirements of fast data insertion, fast query execution, and/or security against information leakages (see Table 1 for a comparison).

Leakages. Cryptographic techniques show following leakages:

(1) *Data distribution leakage from the storage [7]*: allows an adversary to learn the frequency-count of each value by just observing ciphertext. DET reveals such information.

(2) *Search- and access-patterns leakages [7, 16]*: occur during query execution. Search-pattern leakages allow an adversary to learn if and when a query is executed, while access-patterns leakage allows learning which tuples are retrieved (by observing the (physical) address/location of encrypted tuples) to answer a query. Practical techniques (*e.g.*, order-preserving encryption (OPE) [2], DET, symmetric searchable encryption (SSE) [22, 23], and secure hardware-based techniques [3, 33, 42]) reveal access-patterns. In contrast, non-efficient techniques (*e.g.*, secret-sharing [4, 6] or oblivious RAM (ORAM) based techniques) hide access-patterns.

(3) *Volume/output-size leakage [7]*: allows an adversary (having some background knowledge) can deduce the data by simply observing the size of outputs (or the number of qualifying tuples). [7, 16, 26] showed that output-size may also leak data distribution. *Access-patterns revealing techniques implicitly disclose the output-size.* Moreover, the seminal work [19] showed that the output-size revealed even due to access-pattern hiding techniques enables the attacker to reconstruct the dataset. A possible solution is adding fake tuples with the real data, thereby each value has

an identical number of tuples and using indexable SSEs. However, [26] showed that it will be even more expensive than simply scanning the entire database in SGX (or download the data at $\mathcal{DP}$ to execute the query locally). Existing output-size preventing solutions, *e.g.*, Kamara et al. [18] or Patel et al. [31], suffer from one major problem: [18] fetches $\alpha \times max$, $\alpha > 2$, rows, while [31] fetches $2 \times max$ rows with additional secure storage of some rows (which is the function of DB size), where max is the maximum number of rows a value can have. Thus, both [18] and [31] fetch more than the desired rows, *i.e.*, max. Moreover, both [18] and [31] cannot deal with dynamic data.

Existing techniques in terms of data insertion, query execution, and leakages. Existing encrypted search techniques differ in their support for dynamic data, efficient query execution, and offered security properties. For instance, DET supports very efficient insertion and query processing, while its ciphertext data leaks data distribution.

Non-indexable techniques/systems (*e.g.*, SSE [11, 35], secret-sharing (SS) [4, 6], secure hardware-based systems [42]) allow fast data insertions by just encrypting the data, but have inefficient query response time, due to unavailability of an index, and hence, reading the entire data. SS hides search- and access-patterns, while others reveal. Moreover, all such techniques are prone to output-size leakage.

In contrast, indexable techniques/systems (*e.g.*, indexable SSEs (such as PB-Tree [22], IB-Tree [23]) and secure hardware-based index [25]) have faster query execution, but show slow data insertion rate, due to building the entire index at the trusted side for each data insertion; *e.g.*, [30] showed that creating a secure index over 100M rows took more than 1 hour. Moreover, these indexable techniques use ***specialized indexes*** that require specialized encryption and tree traversal protocols that are not supported in the existing standard database systems. This, in turn, limits their usability in dealing with large-scale time-series datasets. All such indexable solutions reveal output-size. While indexable solutions mixed with ORAM (*e.g.*, [30]) hide search- and access-patterns, they are not efficient for query processing (due to several rounds of interaction between the data owner and the server to answer a query). In summary, spatial time-series data adds complexity since (*i*) it can be very large, and (*ii*) arrives dynamically (possibly a high velocity). Existing techniques, as discussed above, are not suitable to support secure data processing over such data.

Advantages of Concealer. (*i*) *Frequent data insert.* We deal with frequent bulk data insertions (which is a requirement of spatial time-series datasets). (*ii*) *Deal with large-size data.* We handle large-sized data with several attributes and large-sized domain efficiently, as our experimental results will show in §8. (*iii*) *Output-size prevention.* While Concealer satisfies the standard security notion (supported by existing SSEs), *i.e.*, indistinguishability under chosen keyword attacks (IND-CKA) [11], it also prevents output-size attacks, unlike IND-CKA. (*iv*) *Oblivious processing in SGX.* As we use the current SGX architecture, suffering from side-channel attacks (*e.g.*, cache-line, branch shadow, and page-fault attack [20, 39, 40]) that enable the adversary to deduce information based on access-patterns in SGX. Thus, we incorporate techniques to deal with these attacks.

1.2 Scoping the Problem

There are other aspects, for them either solutions exist or this paper does not deal with them, as: (*i*) *Key management.* We do not focus on building/improving key infrastructure for public/private keys, as well as, key generation and sharing between

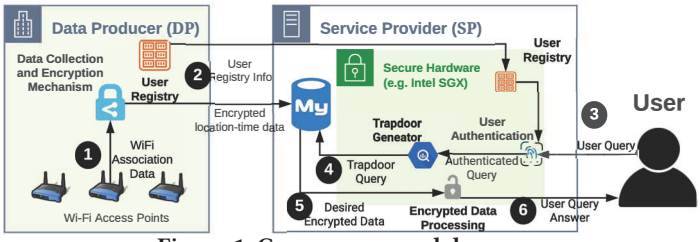


Figure 1: Concealer model.

SGX and $\mathcal{DP}$. Further, changing the keys of encrypted data and re-encrypting the data is out of the scope of this paper, though one may use the recent approach [17] to do so. Also, we do not focus on SGX remote attestation. (*ii*) *Man-in-the-middle (MiM) or replay attacks.* There could be a possibility of MiM and replay attacks on SGX during attestation and query execution. We do not deal with both issues, and techniques [13] can be used to avoid such attacks. (*iii*) *Inference from the number of rows.* Since we send data in epochs, different numbers of tuples in different epochs (*e.g.*, epochs for day vs night time) may reveal information about the user. This can be prevented by sending the same number of rows in each epoch (equals to the maximum rows in any epoch). The current implementation of Concealer does not deal with this issue. (*iv*) *Inference from occupancy count.* Occupancy information mixed with background knowledge reveals the presence/absence of a person at a location (*e.g.*, offices). We do not deal with these inferences, and differential privacy techniques mixed with SGX [41] can be used to deal with such issues.

2 CONCEALER OVERVIEW

This section provides an overview of entities involved in Concealer and its architecture with an overview of algorithms.

2.1 Entities and Assumptions

Concealer consists of the following three entities:

- **Data provider $\mathcal{DP}$:** is a trusted entity that collects user's spatial time-series data as part of its regular operation (*e.g.*, providing cellular service to users). $\mathcal{DP}$ shares such data in encrypted form with service providers $\mathcal{SP}$. $\mathcal{DP}$, also, maintains a *registry*, one per $\mathcal{SP}$, that contains a list of identification information of users, who have registered to use the application provided by the corresponding $\mathcal{SP}$ (*i.e.*, can run queries at that $\mathcal{SP}$). As will be clear, this registry helps to restrict the users to request individualized applications about other users.
- **Service provider $\mathcal{SP}$:** is an untrusted entity that develops location-based applications (as mentioned in §1) over encrypted data. To do so, $\mathcal{SP}$ hosts secure hardware, SGX, that works as a trusted agent of $\mathcal{DP}$.[2] SGX and $\mathcal{DP}$ share a secret key s_k (used for encryption/ decryption of data), and the key s_k is unknown to all other entities.

  An untrusted $\mathcal{SP}$ may try to learn user's data passively by either observing the data retrieved by SGX or exploiting side-channel attacks on SGX during query execution. $\mathcal{SP}$ may, further, learn user's data by actively injecting the fake data into the database and then observing the corresponding ciphertext and query access-patterns. We assume that $\mathcal{SP}$ knows background information, *e.g.*, metadata, the schema of the relation, the number of tuples, and the domain of attributes. However, the adversarial $\mathcal{SP}$ cannot alter anything within the secure hardware and cannot decrypt the data, due to the unavailability of the encryption key. Such assumptions are similar to those considered in the past

[2] The assumption of secure hardware at untrusted third-party machines is consistent with emerging system architectures; *e.g.*, Intel machines are equipped with SGX (see https://newsroom.intel.com/newsroom/wp-content/uploads/sites/11/2017/09/8th-gen-intel-core-product-brief.pdf).

	$\mathcal{L}$	$\mathcal{T}$	$\mathcal{O}$
r_1	l_1	t_1	o_1
r_2	l_1	t_2	o_2
r_3	l_2	t_3	o_2
r_4	l_1	t_4	o_1
r_5	l_2	t_5	o_3
r_6	l_3	t_6	o_2

(a) A relation R in cleartext at $\mathcal{DP}$.

	l_1, l_3	l_2
$t_4\ t_5, t_6$	$cid_1^{\{1,1\}} = \{r_4, r_6\}$	$cid_2^{\{1,2\}} = \{r_5\}$
t_1, t_2, t_3	$cid_1^{\{2,1\}} = \{r_1, r_2\}$	$cid_3^{\{2,2\}} = \{r_3\}$

$cell\_id[] = \{cid_1, cid_2, cid_1, cid_3\}$ $\quad c\_tuple[] = \{4, 1, 1\}$

(b) The grid created at $\mathcal{DP}$ for rows of Table 2a.

	$\mathcal{L}$	$\mathcal{O}$	Tuple	Index($\mathcal{L}, \mathcal{T}$)
r'_1	$\mathcal{E}_k(l_1 \| \| t_1)$	$\mathcal{E}_k(o_1 \| \| t_1)$	$\mathcal{E}_k(l_1 \| \| t_1 \| \| o_1)$	$\mathcal{E}_k(cid_1 \| \| 1)$
r'_7	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_k(f \| \| 1)$
r'_2	$\mathcal{E}_k(l_1 \| \| t_2)$	$\mathcal{E}_k(o_2 \| \| t_2)$	$\mathcal{E}_k(l_1 \| \| t_2 \| \| o_2)$	$\mathcal{E}_k(cid_1 \| \| 2)$
r'_3	$\mathcal{E}_k(l_2 \| \| t_3)$	$\mathcal{E}_k(o_2 \| \| t_3)$	$\mathcal{E}_k(l_2 \| \| t_3 \| \| o_2)$	$\mathcal{E}_k(cid_3 \| \| 1)$
r_8	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_{nd}(fake)$	$\mathcal{E}_k(f \| \| 2)$
r'_4	$\mathcal{E}_k(l_1 \| \| t_4)$	$\mathcal{E}_k(o_1 \| \| t_4)$	$\mathcal{E}_k(l_1 \| \| t_4 \| \| o_1)$	$\mathcal{E}_k(cid_1 \| \| 3)$
r'_5	$\mathcal{E}_k(l_2 \| \| t_5)$	$\mathcal{E}_k(o_3 \| \| t_5)$	$\mathcal{E}_k(l_2 \| \| t_5 \| \| o_3)$	$\mathcal{E}_k(cid_2 \| \| 1)$
r'_6	$\mathcal{E}_k(l_3 \| \| t_6)$	$\mathcal{E}_k(o_2 \| \| t_6)$	$\mathcal{E}_k(l_3 \| \| t_6 \| \| o_2)$	$\mathcal{E}_k(cid_1 \| \| 4)$

$Ecell\_id[2,2] = \mathcal{E}_{nd}(\{cid_1, cid_2, cid_1, cid_3\})$

$Ec\_tuple[3] = \mathcal{E}_{nd}(\{4, 1, 1\})$

(c) Encrypted data with encrypted counters at $\mathcal{SP}$.

Table 2: Input time-series relation and output of data encryption algorithm.

work related to SGX-based computation [33, 42], work on attacks based on background knowledge in [19, 27].

- **User or data consumer $\mathcal{U}$:** that uses the services of $\mathcal{DP}$ (such as cellular or WiFi connectivity) and queries to $\mathcal{SP}$. We assume that $\mathcal{U}$ have their public and private keys, which are used to authenticate $\mathcal{U}$ at $\mathcal{SP}$ (via SGX against registry). As mentioned in §1, $\mathcal{U}$ can request both aggregate and individualized queries. While $\mathcal{U}$ is trusted with the data that corresponds to themselves, they are not trusted with data belonging to other users. Finally, we assume that while $\mathcal{U}$ can execute the aggregation queries, they do not collude with $\mathcal{SP}$, *i.e.*, they do not share cleartext results of any query with $\mathcal{SP}$.

2.2 Architecture

Concealer (Figure 1) consists of the following phases:

Phase 0: *Preliminary step: Announcement of $\mathcal{SP}$ by $\mathcal{DP}$.* As a new $\mathcal{SP}$ is added into the system, $\mathcal{DP}$ announces about the $\mathcal{SP}$ to all their users. Only interested users inform to $\mathcal{DP}$ if they want to use $\mathcal{SP}$'s application. Information of such users, their device-ids, and authentication information is stored by $\mathcal{DP}$ in the registry.

Phase 1: *Data upload by $\mathcal{DP}$.* $\mathcal{DP}$ collects spatial time-series data of the form $\langle l_i, t_i, o_i \rangle$ (❶), where l_i is the location, t_i is the time, and o_i is the observed value at l_i and t_i. For instance, in the case of WiFi data, the location may correspond to the region covered by a specific WiFi access-point, and the observation corresponds to a particular device-id connected to that access-point at a given time. $\mathcal{DP}$ encrypts the data using the mechanism given in §3 and provides the encrypted data to $\mathcal{SP}$ (❷) along with encrypted registry and verifiable tags (to verify the data integrity at $\mathcal{SP}$ by SGX).

Concealer considers the data as a relation R with three attributes: $\mathcal{T}$ (time), $\mathcal{L}$ (location), and $\mathcal{O}$ (observation). Table 2a shows an example of the cleartext spatial time-series data, (which will be used in this paper to explain Concealer). In Table 2a, we have added a row-id r_i ($1 \leq i \leq 6$) to refer to individual rows of Table 2a. Table 2c shows an example of the encrypted spatial time-series data as the output of Concealer.

Phase 2: *Query generation at $\mathcal{U}$.* A query $Q = \langle qa, att \rangle$, where qa is an aggregation (count, maximum, minimum, top-k, and average) or selection operation for a given condition, and att is a set of attributes with predicates on which query will be executed, is submitted to $\mathcal{SP}$ (❸). qa is always encrypted to prevent $\mathcal{SP}$ to know the query values.

Phase 3: *Query processing at $\mathcal{SP}$.* $\mathcal{SP}$ holds encrypted spatial time-series dataset and the user query (submitted to the secure hardware SGX). SGX, first, authenticate the user, and then, translates the query into a set of appropriate secured query trapdoors to fetch the tuples from the databases (❹). Note that since the individualized application is executed for the user itself, trapdoors are only generated if the authentication process succeeds to find that the user is wishing to know his past behavior.

The trapdoors are generated by using the methods of §4 for point queries or of §5 for range queries. On receiving encrypted tuples from the database (❺), the secure hardware, first, *optionally* checks their integrity using verifiable tags, and if find they have not tampered, decrypts them (if necessary), obliviously processes them, and produces the final answer to users (❻).

Phase 4: *Answer decryption at $\mathcal{U}$.* On receiving the answer, $\mathcal{U}$ decrypts them.

2.3 Algorithm Overview

Before going into details of Concealer's data encryption and query execution algorithms, we, first, provide their overview.

Data encryption method at $\mathcal{DP}$: partitions the time into slots, called *epochs*, and for each epoch, it executes the encryption method that consists of the following three stages:

Stage 1: *Setup.* Assume that we want to deal with two attributes (A and B), (*e.g.*, location and time). This stage: (*i*) creates a grid of size, say $x \times y$, (*ii*) sub-partitions the time into y subintervals, *e.g.*, for an epoch of 9-10am, creates y subintervals as: 9:00-9:10, 9:11-9:20, and so on, and (*iii*) using a hash function, say $\mathbb{H}$, allocates x values of A attributes over x columns, allocates y values (or y subintervals) of B attribute to y rows, and allocates some ***cell-ids*** $< x \times y$ (each with their ***counters*** initialized to zero) over the grid cells. (Such grid-creation steps can be used for more than two columns trivially and extended for non-time-series dataset.)

Stage 2: *Encryption:* In this stage, each sensor reading is encrypted and a *verifiable tag* is produced for integrity verification, as: (*i*) a tuple t_i is allocated to a grid cell corresponding to its desired column (*e.g.*, location and time) values using a *hash function*, the counter value of the cell-id is increased by one and attached with the tuples, and the tuples is encrypted to produce secure ciphertext with the encrypted counter value as a new attribute value, (*ii*) a hash-chain is created over the encrypted tuple values of the same cell-id for integrity verification (and verify false data injection or data deletion by $\mathcal{SP}$, and (*iii*) encrypted fake tuples are added (to prevent output-size leakage at $\mathcal{SP}$).

Stage 3: *Sharing:* This stage sends encrypted real and fake tuples with encrypted verifiable tags and encrypted cell-id, counter information to $\mathcal{SP}$.

Example. Table 2a shows six cleartext rows of an epoch. A 2×2 grid with three cell-ids cid_1, cid_2, and cid_3 is shown in Table 2b. Six cleartext rows are distributed over different cells of the grid. Table 2c shows the output of the encryption algorithm with fake tuples to prevent the output-size attack at $\mathcal{SP}$ and an index column created over the cell-ids. Encrypted Table 2c with counters and cell-ids (written below Table 2b) in encrypted form is given to $\mathcal{SP}$. ***Details of the encryption method and example will be given in §3.***■

Data insertion into DBMS at $\mathcal{SP}$: On receiving encrypted data from $\mathcal{DP}$, $\mathcal{SP}$ inserts the data into DBMS that creates/modifies the *index based on the counters* associated with each tuple.

Query execution at $\mathcal{SP}$: As a pre-processing phase, the enclave at $\mathcal{SP}$, first, authenticates the user, as mentioned in Phase 3 of §2.3, and then, executes the query, as follows:

Point queries. Consider a query on a location l and time t. For answering this, the enclave at $\mathcal{SP}$ executes the following steps:

(i) first execute the hash function $\mathbb{H}$ on query predicate l and t to know the cell-id, say cid_z, that was allocated by $\mathcal{DP}$ to l and t, (ii) using the information of cell-id and counter information, which was sent by $\mathcal{DP}$, create *static bins of a fixed size* (to prevent output-size leakage), (iii) among the created bins, find a bin, say B_i that has the cell-id cid_z that was obtained in the first step above, and (iv) fetch data from DBMS corresponding to the bin B_i, and (v) verify the integrity of data (if needed), *obliviously* process the data against query predicate in the enclave, and decrypt only the desired data.

Range queries. A range query, of course, can be executed by using the above point query method by converting the range query into several point queries, but will incur the overhead. To avoid the overhead of several point queries, we create static bins of fixed size over the fixed-sized groups of subintervals and fetch such bins to answer the query by following point queries' step (v). Following ***§3,§4,§5 will describe these algorithms in details***, and then §8 will compare these algorithms on different datasets and against different systems.

Example. Underlying DBMS at $\mathcal{SP}$ creates an index over Index column of Table 2c. SGX creates two bins over cell-ids's as $B_1 : \langle cid_1 \rangle$, $B_2 : \langle cid_2, f||1, f||2 \rangle$. Note that both bins corresponds to four rows—B_1 will fetch r_1, r_2, r_4, r_6, and B_2 will fetch $r_3, r_5, f||1, f||2$ rows. Thus, the output size will be the same. Now, consider a query $Q = \langle \text{count}, (l_2, t_5) \rangle$ over Table 2c. Here, SGX will know that it needs to fetch rows corresponding to the bin having cid_2, by generating four trapdoors: $\mathcal{E}_k(cid_2||1)$, $\mathcal{E}_k(cid_3||1)$, $\mathcal{E}_k(f||1)$, and $\mathcal{E}_k(f||2)$. Finally, based on the retrieve rows, SGX produces the final answer.■

3 DATA ENCRYPTION AT DATA PROVIDER

Concealer stores data in discredited time slots, called *epochs* or *rounds*. Epoch duration is selected based upon the latency requirements of $\mathcal{SP}$. Executing queries over multiple epochs could lead to inference attacks, and for dealing with it, we will present a method in §6. This section describes Algorithm 1, which is executed at $\mathcal{DP}$, for encrypting time-series data (assuming with three attributes location $\mathcal{L}$, time $\mathcal{T}$, and object O) belonging to one epoch. (In our experiments §8, we will consider different datasets with multiple columns.) Algorithm 1 uses deterministic encryption (DET) to support fast query execution. Since DET produces the same ciphertext for more than one occurrence of the same location and object, to ensure ciphertext indistinguishability, we concatenate each occurrence of the location and observation values with the corresponding timestamp.

In Concealer, queries retrieve a subset of tuples based on predicates specified over attributes, such as $\mathcal{L}$, O, or both. Queries, further, are always associated with ranges over time (see Table 4). Thus, to support the efficient execution of such queries, Concealer creates a cell-based index over query attributes (*e.g.*, $\mathcal{L}$ and/or O) along with time. For simplicity, Algorithm 1 illustrates how a cell-based index is created for location and time attributes, Index($\mathcal{L}, \mathcal{T}$). Similar indexes can also be created for other attributes, such as Index($O, \mathcal{T}$) and Index($\mathcal{L}, O, \mathcal{T}$). Details of Algorithm 1 is given below:

Key generation (Lines 2). Since using a single key over multiple epochs results in the identical ciphertext of a value, Concealer produces a key for encryption for each epoch, as: $k \leftarrow s_k||eid$, where s_k is the secret key shared between SGX and $\mathcal{DP}$, eid is the epoch-id (which is the starting timestamp of the epoch), and $||$ denotes concatenation. Thus, encrypting a value v using k in two different epochs will produce different ciphertexts. (Only the first eid and epoch duration is provided to SGX to generate other eids to decrypt the data during query execution.)

Algorithm 1: Data encryption algorithm.

```
Inputs: R: a relation. ℍ: a hash function. ℰ(): an encryption function. s_k: secret key.
Outputs: E(R): the encrypted relation.
1  Variables: ∀c_t ← 0, where 1 ≤ t ≤ r. x ← #ℍ(Domain(ℒ)),
     y ← #ℍ(Domain(𝒯)), cell_id[x, y] ← 0, c_tuple[u] ← 0.
2  Function key_gen(s_k) begin
3      k ← (s_k||eid)
4  Function encrypt_data(R) begin
5      for j ∈ (0, n − 1) do
6          Eo_j ← ℰ_k(o_j||t_j), El_j ← ℰ_k(l_j||t_j), Er_j ← ℰ_k(v_j||l_j||t_j)
7          Function Cell-Formation(j^th tuple) begin
8              p ← ℍ(l_j), q ← ℍ(t_j), cid_z^{p,q} ← cell_id[p, q]
9              c_t ← c_tuple[cid_z^{p,q}] ← c_tuple[cid_z^{p,q}] + 1
10             Ec_j ← E_k(cid_z^{p,q} ||c_t)
11         return E(R) ← ⟨Eo_j, El_j, Er_j, Ec_j⟩
12 Function add_fake_tuples() begin
13     for j ∈ (0, n − 1) do
14         Generate fake Eo_j, El_j, and Er_j, and Ec_j ← ℰ_k(f||j)
15         Append the j^th fake tuple to the relation E(R)
16 Function HashChain(c_tuple[u], ⟨Eo, El, Er⟩) begin
17     for j ∈ c_tuple[ ], ∀p tuples with same cell-id do
18         h_l^j ← H(El_p)||(H(El_{p−1})||(...||(H(El_2)||H(El_1))) ...)))
19         h_o^j ← H(Eo_p)||(H(Eo_{p−1})||(...||(H(Eo_2)||H(Eo_1))) ...)))
20         h_r^j ← H(Er_p)||(H(Er_{p−1})||(...||(H(Er_2)||H(Er_1))) ...)))
21         Ehl^j ← E(h_l^j), Eho^j ← E(o_l^j), Ehr^j ← E(h_r^j)
22 Function Transmit(E(R), cell_id[x, y], c_tuple[u]) begin
23     Ecell_id[x, y] ← ℰ_nd(cell_id[x, y]), Ec_tuple[u] ← ℰ_nd(c_tuple[u])
24     Permute all the tuples of the encrypted relation E(R)
25     Send E(R), Ecell_id[x, y], Ec_tuple[u], Ehl^j, Eho^j, Ehr^j
```

Tuple encryption (Lines 4-11). As the tuple arrives, it got appropriately encrypted (Line 6) using DET. Note that by encryption over the concatenated time with location and object values, results in a unique value in the entire relation. Now, in order to allocate the cell value to be used as the index, we proceed as follows: Let $|\mathcal{L}|$ be the number of locations and $|\mathcal{T}|$ be the duration of the epoch. Concealer maps the set of location $|\mathcal{L}|$ into a range of values from 1 to $x \leq |\mathcal{L}|$ using a simple hash function. It, furthermore, partitions $|\mathcal{T}|$ into $y > 1$ subintervals of duration, $|\mathcal{T}|/y$, which are then mapped using a hash function into $y > 1$ values. Thus, all tuples of the epoch are distributed randomly over the grid of $x \times y$ (see Example 3 below). Then, u cell-ids ($u < x \times y$) are allocated to grid cells. To refer to the cell-id of a cell, we use the notation $cid_z^{\{p,q\}}$ that shows that the cell $\{p, q\}$ is assigned a cell-id cid_z. In Step 3 of query execution §4.2, it will be clear that we will fetch tuples to answer any query based on cell-ids, instead of directly using query predicates.

Here, we keep two vectors: (i) *cell_id* of length $x \times y$ to keep the cell-id allocated to each cell of the grid, and (ii) *c_tuple* of length u to store the number of tuples that have been allocated the same cell-id. During processing a j^{th} tuple, we increment the current counter value of the number of tuples that have the same cell-id by one using *c_tuple*, and encrypts it. This value will be allocated to Index($\mathcal{L}, \mathcal{T}$)$\rangle$ attribute of the j^{th} (Lines 9-10).

Allocating fake tuples (Lines 12 -15). Since $\mathcal{SP}$ will read the data from DBMS into the enclave, different numbers of rows according to different queries may reveal information about the encrypted data. Thus, to fetch an equal number of rows for any query, $\mathcal{DP}$ needs to share some fake rows. There are two methods for adding the fake rows:

(i) *Equal number of real and fake rows*: This is the simplest method for adding the fake rows. Here, $\mathcal{DP}$ adds ciphertext secure fake tuples. The reason of adding the same number of real and fake rows is dependent on the property of the bin-packing algorithm,

which we will explain in §4.2 (Theorem 4.1).[3] In Index attribute, a j^{th} fake tuple contains an encrypted identifier with the tuple-id j, denoted by $\mathcal{E}_k(f||j)$, where f is an identifier (known to only $\mathcal{DP}$) to distinguish real and fake tuples.

(*ii*) *By simulating the bin-creation method*: To reduce the number of fake rows to be sent, we use this method in which $\mathcal{DP}$ simulates the bin-packing algorithm (as will be explained in §4.2) and finds the total number of fake rows required in all bins such that their sizes must be identical. Then, $\mathcal{DP}$ share such ciphertext secure fake tuples with their Index values, as in the previous method. As will be clear soon by Theorem 4.1 in §4.1, in the worst case, both the fake tuple generation methods send the same number of fake tuples, *i.e.*, an equal number of real and fake tuples.

Hash-chain creations (an optional step) Line 16-21. $\mathcal{DP}$ creates hash chains over encrypted tuples allocated the same cell-id, as follows: let p be the numbers of tuples with the same cell-ids and p encrypted location ciphertext as: $E(l_1), E(l_2), \ldots, E(l_p)$. Now, $\mathcal{DP}$ executes a hash function as follows:

$$h_{l1} \leftarrow H(E(l_1))$$
$$h_{l2} \leftarrow H(E(l_2)||h_{l1})$$
$$\ldots$$
$$h_l \leftarrow H(E(l_p)||h_{l(p-1)})$$

In the same way, hash digests for other columns are computed, and the final hash digest (*i.e.*, h_l) is encrypted that works as a verifiable tag at $\mathcal{SP}$.

Sending data (Line 22-25). Finally, $\mathcal{DP}$ permutes all encrypted tuples of the epoch to mix fake and real tuples in the relation and sends them with the two encrypted vectors *Ecell_id*[] and *Ec_tuple*[] and encrypted hash digests.[4]

Example 3. Now, we explain with the help of an example how encryption algorithm works. Consider six rows of Table 2a as the rows of an epoch, and we wish to encrypt those tuples with index on attributes $\mathcal{L}$ and $\mathcal{T}$. Assume that Algorithm 1 creates a 2×2 grid (see Table 2b) with three cell-ids: cid_1, cid_2, and cid_3. Table 2b shows two vectors *cell_id*[] and *c_tuple*[] corresponding to $\mathcal{L}$ and $\mathcal{T}$ attributes. Values in *c_tuple*[] show that the number of tuples has been allocated the same cell-id. For instance, $c\_tuple[1] = 4$ shows that four tuples are allocated the same cell-id (*i.e.*, cid_1). In Table 2b, for explanation purposes, we show which rows of Table 2a correspond to which cell; however, this information is not stored, only information of vectors *cell_id*[] and *c_tuple*[] is stored.

The complete output of Algorithm 1 is shown in Table 2c for cleartext data shown in Table 2a, where Index($\mathcal{L}, \mathcal{T}$) is the column on which DBMS creates an index. In Table 2c, $\mathcal{E}$ refers to DET, $\mathcal{E}_{nd}$ refers to a non-deterministic encryption function, and k be the key used to encrypt the data of the epoch. In addition, we create three hash chains, one hash chain per cell-id. Also, note that this example needs only 2 fake tuples to prevent output-size leakage at $\mathcal{SP}$.■

4 POINT QUERY EXECUTION

This section develops a bin-packing-based (BPB) method for executing point queries. Later, §5 will develop a method for range queries. The objectives of BPB method are twofold: first, create identical-size bins to prevent leakages due to output-size, (*i.e.*, when reading some parts of the data from disk to the enclave), and second, show that the addition of *at most n* fake tuples is enough in the worst case to prevent output-size leakage, where n is the number of real tuples. BPB method partitions the values of *c_tuple*[] into almost equal-sized bins, using which a query can be executed. Note that bins are created only once, prior to the first query execution. This section, first, presents the bin-creation method, and then, BPB query execution method.

4.1 Bin Creation

Bins are created inside the enclave using a bin-packing algorithm, after decrypting vector *Ec_tuple*[].

Bin-packing algorithms. A bin-packing algorithm places the given inputs having different sizes to bins of size at least as big as the size of the largest input, without partitioning an input, while tries to use the minimum number of bins. First-Fit Decreasing (FFD) and Best-Fit Decreasing (BFD) [8] are the most notable bin-packing algorithms and *ensure that all the bins (except only one bin) are at least half-full.*

In our context, u cell-ids ($cid_1, cid_2, \ldots, cid_u$) are inputs to a bin-packing algorithm, and the number of tuples having the same cell-id is considered as a weight of the input. Let *max* be the maximum number of tuples having the same cell-id cid_i. Thus, we create bins of size at least $|b| = max$ and execute FFD or BFD over u different cell-ids, resulting in $|Bin|$ bins as an output of the bin-packing algorithm.

The minimum number of bins. Let n be the number of real tuples sent by $\mathcal{DP}$, *i.e.*, $n = \sum_{i=1}^{i=u} c\_tuple[i]$. Let $|b|$ be the size of each bin. Thus, it is required to divide n inputs into at least $\lceil n/|b| \rceil$ bins.

Theorem 4.1. (Upper bounds on the number of bins and fake tuples) *The above bin-packing method using a bin size $|b|$ achieves the following upper bounds: the number of bins and the number of fake tuples sent by $\mathcal{DP}$ are at most $\frac{2n}{|b|}$ and at most $n + \frac{|b|}{2}$, respectively, where $n \gg |b|$ is the number of real tuples sent by $\mathcal{DP}$.*

In the full version (https://arxiv.org/abs/2102.05238), we will provide the proof of this theorem.

Equi-sized bins. The bins produced by FFD/BFD may have different numbers of tuples. Thus, we pad each bin with fake tuples, thereby all bins have $|b|$ tuples. Let $tuple_{b_i} < |b|$ be the number of tuples assigned to an i^{th} bin (denoted by b_i). Here, the ids (*i.e.*, the value of Index column) of fake tuples allocated to the bin b_i will be $|b| - tuple_{b_i}$, and all these fake tuple ids cannot be used for padding in any other bin. Thus, for padding, we create disjoint sets of fake tuple ids (see the example below to understand the reason).

Example 4.1. Assume five cell-ids $cid_1, cid_2, \ldots, cid_5$ having the following number of tuples $c\_tuple[5] = \{79, 2, 73, 7, 7\}$. Here, cid_1 has the maximum number of tuples; hence, the bin-size is at least 79. After executing FFD bin-packing algorithm, we obtain three bins, each of size 79: b_1: $\langle cid_1 \rangle$, b_2: $\langle cid_3, cid_2 \rangle$, and b_3: $\langle cid_5, cid_4 \rangle$. Here, bins b_2 and b_3 needs 4 and 65 fake tuples, respectively. One can think of sending only 65 fake tuples to access bins b_2 and b_3 to have size 79. However, in the absence of access-pattern hiding techniques, the adversary will observe that any 4 tuples out of 65 fake tuples are accessed in both bins. It will reveal that these four tuples are surely fake, and thus, the adversary may deduce that the bin size of b_2 is 75. Thus, $\mathcal{DP}$ needs to send 69 fake tuples in this example.■

4.2 Bin-Packing-based Query Execution

We develop bin-packing-based (BPB) method (see pseudocode in Algorithm 2) based on the created bins (over location and time

[3] For n real tuples, we add a little bit more than n fake tuples in the worst case (Theorem 4.1).
[4] The size of both vectors is significantly smaller (see experimental section §8.1).

Algorithm 2: Bin-packing-based query execution method.

Inputs: $\langle qa, (\mathcal{L} = l, \mathcal{T} = t)\rangle$: a query qa involving predicates over $\mathcal{L}$ and $\mathcal{T}$ attributes. $cell\_id[x, y]$, $c\_tuple[u]$, $\mathbb{H}$: A hash function, $\mathcal{E}_k()$: An encryption function using a key k.
$|Bin|$: the number of bins. $b[i][j]$: i^{th} bin having j cell-ids, where $j > 0$.
Outputs: A set of ciphertext queries.

1 **Function** *Query_Execution*$(\langle qa, (\mathcal{L} = l, \mathcal{T} = t)\rangle)$ **begin**
2 **Function** *Find_cell*(l, t) **begin**
3 $p \leftarrow \mathbb{H}(l)$, $q \leftarrow \mathbb{H}(t)$, $cid_z \leftarrow cell\_id[p, q]$
4 **return** cid_z; **break**
5 **Function** *Find_bin*$(cid_z, |Bin|, b[*][*])$ **begin**
6 **for** $desired \in (0, |Bin| - 1)$ **do**
7 **if** $cid_z \in b[desired][*]$ **then**
8 **return** $b[desired][*]$; **break**
9 **Function** *Formulate_queries*$(b[desired][*])$ **begin**
10 **for** $\forall j \in (b[desired][j])$ **do**
11 $cell\_id \leftarrow b[desired][j]$, $counter \leftarrow c\_tuple[cell\_id]$
12 $\forall counter$, generate ciphertexts $E_k(cell\_id||counter)$

attributes). A similar method can be extended for other attributes. BPB method contains the following four steps:

Step 0: Bin-creation. By following FFD or BFD as described above, this step creates bins over cell-ids ($c\_tuple[]$), if bins do not exist.

Step 1: Cell identification (Lines 2-4). The objective of this step is to find a cell of the grid corresponding to the requested location and time. A query $Q_e = \langle qa, (\mathcal{L} = l, \mathcal{T} = t)\rangle$ is submitted to the enclave that, on the query predicates l and t, applies the hash function $\mathbb{H}$, which was also used by $\mathcal{DP}$ (in *Cell-Formation* function, Line 8 of Algorithm 1). Thus, the enclave knows the cell, say $\{p, q\}$, corresponds to l and t. Based on the cell $\{p, q\}$ and using the vector $cell\_id[]$, it knows the cell-id, say cid_z, allocated to the cell $\{p, q\}$.

Step 2: Bin identification (Lines 5-8). Based on the output of Step 1, *i.e.*, the cell-id cid_z, this step finds a bin b_i that contains cid_z. Bin b_i may contain several other cell-ids along with identities of the first and the last fake tuples required for b_i.

Step 3: Query formulation (Lines 9-12). After knowing all cell-ids that are required to be fetched for bin b_i, the enclave formulates appropriate ciphertexts that are used as queries. Let the set of cell-ids in b_i be $C_1, C_2, \ldots, C_\alpha$, containing $\#_1, \#_2, \ldots, \#_\alpha$ records, respectively. Let the fake tuple range for b_i be f_l and f_h (let $\#_f = f_h - f_l$ be the number of fake tuples that have to be retrieved for b_i). The enclave generates $\#_i$ number of queries, as: $\mathcal{E}_k(C_p||j)$, where $1 \leq j \leq \#_i$ for each cell C_p corresponding to b_i and k is the key obtained by concatenating s_k and epoch-id (as mentioned in Line 2 of Algorithm 1). Also, it generates $\#_f$ fake queries, one for each of the fake tuples associated with b_i.

Advantage of cell-ids. Now, observe that a bin may contain several locations and time values (or any desired attribute value). Fetching data using cell-id does not need to maintain fine-grain information about the number of tuples per location per time.

Step 4: Integrity verification and final answers filtering. We may optionally verify the integrity of the retrieved tuples. To do so, the enclave, first, creates a hash chain over the real encrypted tuples having the same cell-id, by following the same steps as $\mathcal{DP}$ (Lines 16 of Algorithm 1). Then, compares the final hash digest with the decrypted verifiable tag, provided by $\mathcal{DP}$.

Now, to answer the query, the enclave, first, filters those tuples that do not qualify the query predicate, since all tuples of a bin may not correspond to the answer. Thus, decrypting each tuple to check against the query $Q_e = \langle qa, (\mathcal{L} = l, \mathcal{T} = t)\rangle$ may increase the computation cost. To do so, after implementing the above-mentioned Step 3, the enclave generates appropriate filter values ($\mathcal{E}_k(l_i||t_i)$ or $\mathcal{E}_k(o_i||t_i)$, which are identical to the created by $\mathcal{DP}$ using Algorithm 1); while, at the same time, DBMS executes queries on the encrypted data. On receiving encrypted tuples

```
max(int x,int y){
bool getX = ogreator(x, y),
return omove(getX, x, y)
}
```

(a) Oblivious maximum.

```
mov rcx, x
mov rdx, y
cmp rcx, rdx
setg al
retn
```

(b) Oblivious compare: `ogreator`.

```
mov rcx, cond
mov rdx, x
mov rax, y
test rcx, rcx
cmovz rax, rdx
retn
```

(c) Oblivious move: `omove`.

Figure 2: Register-oblivious operators [28].

from DBMS, the enclave performs string-matching operations using filters and decrypts only the desired tuples, if necessary.

Example 4.2. Consider the cells created in Example 3.1, *i.e.*, $cell\_id[] = cid_1, cid_2, cid_1, cid_4$ and $c\_tuple[] = \{4, 1, 1\}$. Now, assume that there are two bins, namely b_1: $\langle cid_1\rangle$ and b_2: $\langle cid_2, cid_3\rangle$. Consider a query $Q = \langle$count, $(l_2, t_5)\rangle$, *i.e.*, find the number of people at location l_2 at time t_5 on the data shown in Table 2c. Here, after implementing Step 1 and Step 2 of BPB method, the enclave knows that cell-id cid_2 satisfies the query, and hence, the tuples corresponding to bin b_2 are required to be fetched. Thus, in Step 3, the enclave generates the following four queries: $\mathcal{E}_k(cid_2||1)$, $\mathcal{E}_k(cid_3||1)$, $\mathcal{E}_k(f||1)$, and $\mathcal{E}_k(f||2)$. Finally, in Step 4, the filtering via string matching is executed over the retrieved four tuples against $\mathcal{E}_k(l_2||t_5)$. Since all the four retrieved tuples have a filter on location and time values, here is no need to decrypt the tuple that does not match the filter $\mathcal{E}_k(l_2||t_5)$.■

4.3 Oblivious Trapdoor Creation & Filtering

Steps for generating trapdoor (Step 3) and answer filtering (Step 4), in §4.2, were not oblivious due to side-channel attacks (*i.e.*, access-patterns revealed via cache-lines and branching operations) on the enclave. Thus, we describe how can the enclave produce queries and process final answers obliviously for preventing side-channel attacks.

Step 3. Let $\#_{Cmax}$ be the maximum cells required to form a bin. Let $\#_{max}$ be the maximum tuples with a cell-id. Let $\#_i$ be the number of tuples with a cell-id C_i. For a bin b_i, the enclave generates $\#_{Cmax} \times \#_{max}$ numbers of queries: $\mathcal{E}_k(C_i||j, v)$, where $1 \leq j \leq \#_{max}$, $C_i \leq \#_{Cmax}$, and $v = 1$ if $j \leq \#_i$ and C_i is required for b_i; otherwise, $v = 0$. Note that *this step produces the same number of queries for each cell and each bin.*

Let $\#_{fmax}$ be the maximum fake tuples required for a bin. Let $\#_{fb_i}$ be the maximum fake tuples required for b_i. The enclave generates $\#_{fmax}$ number of fake queries: $\mathcal{E}_k(f||j, v)$, where $1 \leq j \leq \#_{fmax}$ and $v = 1$ if $j \leq \#_{fb_i}$; otherwise, 0. This step *produces the same number of fake queries for any bin.* Finally, the enclave sorts all real and fake queries based on value v using a data-independent sorting algorithm (*e.g.*, bitonic sort [5]), such that all queries with $v = 1$ precede other queries, and sends only queries with $v = 1$ to the DBMS.

Step 4. The enclave reads all retrieved tuples and appends $v = 1$ to each tuple if they satisfy the query/filter; otherwise, $v = 0$. Particularly, an i^{th} tuple is checked against each filter, and once it matches one of the filters, $v = 1$ remains unchanged; while the value of $v = 1$ is overwritten for remaining filters checking on the i^{th} tuple. It hides that which filter has matched against a tuple. Then, based on v-value, it sorts all tuples using a data-independent algorithm.[5] (After this all tuples with $v = 1$ are decrypted and checked against the query by following the same procedure, if needed.)

Branch-oblivious computation. Note that after either generating an equal number of queries for any bin or filtering the retrieved tuples using a data-oblivious sort, the entire computation is still

[5] If all tuples can reside in the enclave, then bitonic sort is enough. Otherwise, to obliviously sort the tuples, we use column sort [21] instead of the standard external merge sort.

T_4	$cid_1^{\{1,1\}} = 40$	$cid_6^{\{1,2\}} = 30$	$cid_7^{\{1,2\}} = 2$	$cid_{11}^{\{1,4\}} = 9$
T_3	$cid_2^{\{2,1\}} = 50$	$cid_7^{\{2,2\}} = 50$	$cid_6^{\{2,3\}} = 21$	$cid_9^{\{2,4\}} = 60$
T_2	$cid_3^{\{3,1\}} = 60$	$cid_{11}^{\{3,2\}} = 40$	$cid_4^{\{3,3\}} = 45$	$cid_8^{\{3,4\}} = 48$
T_1	$cid_3^{\{4,1\}} = 40$	$cid_{10}^{\{4,2\}} = 50$	$cid_{10}^{\{4,3\}} = 10$	$cid_5^{\{4,4\}} = 50$
	l_1	l_2	l_3	l_4

Table 3: A 4×4 grid.

vulnerable to an adversary that can observe conditional branches, *i.e.*, an if-else statement used in the comparison. To overcome such an attack, we use the idea proposed by [28] that suggested that any computation on registers cannot be observed by an adversary since register contents are not accessible to any code outside of the enclave; thus, register-to-register computation is oblivious. For this, [28] provided two operators: `omove` and `ogreater`, see Figure 2. For any comparison in the enclave, we use these two operators; readers may find additional details in [28].

5 RANGE QUERY EXECUTION

This section develops an algorithm for executing range queries, by modifying BPB method, given in §4.2. For simplicity, we consider a range condition on time attribute. For illustration purposes, this section uses a 4×4 grid (see Table 3, which was created by $\mathcal{DP}$ using Algorithm 1, §3) corresponding to location and time attributes of a relation. In this grid, 11 cell-ids are used, and a number in a cell shows the number of tuples allocated to the cell. The notation T_i shows an i^{th} sub-time interval (after creating a grid using Algorithm 1 §3).

5.1 Enhanced Bin-Packing-Based (eBPB) Method

eBPB method requires $\mathcal{DP}$ to send the number of tuples in each cell of the grid with the vector *cell_id*[]. Thus, it avoids sending the vector *c_tuple*[]. For example, for the grid shown in Table 3, *cell_id*$[4,4] = \{(1,40),(6,30),(7,2),(11,9),(2,50),(7,50),(6,21),(9,60),(3,60),(11,40),(4,45),(8,48),(3,40),(10,50),(10,10),(5,50)\}$. This information helps us in creating bins more efficiently for a range query, as follows:

Step 1: Preliminary step. The enclave decrypts the encrypted vector *Ecell_id*[].

Step 2: Finding top-ℓ cell-ids. Find top-ℓ cells having the maximum number of tuples in one of the locations, where ℓ is the number of cells required to answer the range query. Say, location l_i has top-ℓ cells that have the maximum number of tuples, denoted by *bsize* tuples.

Step 3: Create bins. Execute this step either if ℓ cells required for the current query are more than the cells required for any previously executed query or it is the first query. Fix the bin size to *bsize* and execute FFD that takes $cid_z^{\{p,q\}}$ as inputs and the number of tuples having $cid_z^{\{p,q\}}$ as the weight of the input. If the bin does not have *bsize* number of tuples, add fake tuples to the bin. It results in $|Bin|$ number of bins and then, use all such bins for answering any range covered by ℓ cells.

Step 4: Query formulation and final answers filtering. Find the desired bin satisfying the range query and formulate appropriate queries, as we formed in Step 3 of BPB method §4.2. The DBMS executes all queries and provides the desired tuples to the enclave. The enclave executes the final processing of the query, likewise Step 4 of BPB method §4.2. *Note that for oblivious query formulation and result filtering, we use the same method as described in §4.3.*

Example 5.1.1. Consider a query to count the number of tuples at the location l_1 during a given time interval that is covered by T_2 to T_4. This query spans over three cells; see Table 3. Here, the maximum number of tuples in any three cells at locations l_1, l_2, l_3, and l_4 are $60+50+40 = 150$, $50+50+40 = 140$, $45+21+5 = 71$, and $60+50+48 = 158$, respectively. Thus, the bin of size 158 can satisfy any query that spans over any three cells (arranged in a column) of the grid. ■

Example 5.1.2: *attack on eBPB.* Consider the following queries on data shown in Table 3: (Q_1) retrieve the number of tuples having location l_1 during a given time interval that is covered by T_1 and T_2, and (Q_2) retrieve the number of tuples having location l_1 during a given time interval that is covered by T_2 and T_3. Answering Q_1 and Q_2 may reveal the number of tuples having T_1, T_2, and T_3, as: in answering Q_2 we do not retrieve 40 tuples (corresponding to $\{4,1\}$ cell) that were sent in answering Q_1 and retrieve 50 new tuples (corresponding to $\{2,1\}$ cell). It, also, reveals that 60 tuples (corresponding to $\{3,1\}$ cell) belong to T_2. Note that all such information was not revealed, prior to query execution, due to the ciphertext indistinguishable dataset.■

5.2 Highly Secured Range Query: winSecRange Method

We, briefly, explain a method to prevent the above-mentioned attacks on a range query. Particularly, we fix the length of a range, say $\lambda > 1$, and discretize n domain values, say $v_1, v_2, \ldots, v_n$, into $\lceil \frac{n}{\lambda} \rceil$ intervals (denoted by $\mathcal{I}_i$, $1 \leq i \leq \lceil \frac{n}{\lambda} \rceil$), as: $\mathcal{I}_1 = \{v_1, v_2, \ldots, v_\lambda\}$, …, $\mathcal{I}_{\lceil \frac{n}{\lambda} \rceil} = \{v_{n-\lambda}, \ldots, v_{n-1}, v_n\}$. Here, the bin size equals to the maximum number of tuples belonging to an interval, and bins are created for each interval *only once*. For example, consider 12 domain values: $v_1, v_2, \ldots v_{12}$, and $\lambda = 3$. Thus, we obtain intervals: $\mathcal{I}_1 = \{v_1, v_2, v_3\}$, $\mathcal{I}_2 = \{v_4, v_5, v_6\}$, $\mathcal{I}_3 = \{v_7, v_8, v_9\}$, and $\mathcal{I}_4 = \{v_{10}, v_{11}, v_{12}\}$. Here, four bins are created, each of size equals to the maximum number of tuples in any of the intervals. Now, we can answer a range query of length β by using one of the following methods:

(*i*) $\beta \leq \lambda$ and $\beta \in \mathcal{I}_i$: Here, the entire range exists in $\mathcal{I}_i$; hence, we retrieve only a single entire bin satisfying the range. E.g., if a range is $[v_1, v_2]$, then we need to retrieve the bin corresponding to $\mathcal{I}_1$. (*ii*) $\beta \leq \lambda$ and $\beta \in \{\mathcal{I}_i, \mathcal{I}_j\}$: It may be possible that $\beta \leq \lambda$ but the range β lies in $\mathcal{I}_i$ and $\mathcal{I}_j$, $i \neq j$. Thus, we need to retrieve two bins that cover $\mathcal{I}_i$ and $\mathcal{I}_j$, and hence, we also prevent the attack due to sliding the time window (see Example 5.1.2). E.g., if a range is $[v_2, v_4]$, then we need to retrieve the bins corresponding to $\mathcal{I}_1$ and $\mathcal{I}_2$. (*iii*) $\beta = z \times \lambda$: Here, a range may belong to at most $z+2$ intervals. Thus, we may fetch at most $z+1$ bins satisfying the query. E.g., if a range is $[v_3, v_8]$, then this range is satisfied by intervals $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$; thus, we fetch bins corresponding to $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$.

6 SUPPORTING DYNAMIC INSERTION

Dynamic insertion in Concealer is supported by batching updates into rounds/epochs, similar to [12]. Tuples inserted in an i^{th} period are said to belong to the round rd_i or epoch eid_i. The insertion algorithm is straightforward. Concealer applies Algorithm 1 on tuples of epochs prior to sending them to $\mathcal{SP}$. Note that since Algorithm 1 for distinct rounds is executed independently, the tuples corresponding to the given attribute value (*e.g.*, location-id) may be associated with different bins in different rounds. Retrieving tuples of a given attribute value across different rounds needs to be done carefully, since it might result in leakage, as shown next.

Example 6.1. Consider that the bin size is three, and we have the following four bins for each round of data insertion, where a bin b_i stores tuples of a location l_j:

Queries	Execution (filtering, decryption, and final processing) by the secure hardware
Location and time attributes	
Q1: # observations at l_i during time t_1 to t_x	SM using the filters $El_1 \leftarrow E_k(l_i \| t_1), El_2 \leftarrow \mathcal{E}_k(l_i \| t_2), \ldots, El_k \leftarrow \mathcal{E}_k(l_i \| t_x)$. No decryption needed.
Q2: Locations that have top-k observations during t_1 to t_x	SM using the filters $El_m \leftarrow \mathcal{E}_k(l_i \| t_j)$ where $i \in Domain(\mathcal{L})$ and $j \in \{t_1, t_x\}$, and then, decrypt $\mathcal{E}_k(l \| t \| o)$
Q3: Locations that have at least 10 observations during t_1 to t_x	of qualified tuples only for final processing.
Observation and time attributes	
Q4: Which locations have an observation o_i during t_1 to t_x	SM using $Eo_j \leftarrow \mathcal{E}_k(o_i \| t_j)$, $j \in \{t_1, t_x\}$, and then, decrypt $\mathcal{E}_k(l \| o \| t)$ of qualified tuples to know locations.
Observation, location, and time attributes	
Q5: # an observation o_i has happened at l_i during t_1 to t_x	SM using $Eo_j \leftarrow \mathcal{E}_k(o_i \| t_j \| l_i)$, where $j \in \{t_1, t_x\}$. No decryption needed.

Table 4: Sample queries. *Notation*: SM: String matching.

rd_1 : $b_1 : \langle l_1, l_2, l_3 \rangle$ $b_2 : \langle l_4, l_4, l_4 \rangle$ $b_3 : \langle l_5, l_5, l_5 \rangle$ $b_4 : \langle l_6, l_6, l_6 \rangle$
rd_2 : $b'_1 : \langle l_1, l_1, l_1 \rangle$ $b'_2 : \langle l_2, l_2, l_2 \rangle$ $b'_3 : \langle l_3, l_3, l_3 \rangle$ $b'_4 : \langle l_4, l_5, l_6 \rangle$

Now, answering a query for l_1 fetches bins b_1 and b'_1; a query for l_2 fetches b_1 and b'_2; and a query for l_3 fetches b_1 and b'_3. Here, b_1 is retrieved with three new bins (b'_1, b'_2, b'_3); it reveals that b_1 has three distinct locations. Similarly, b'_4 will be retrieved with three older bins (b_2, b_3, and b_4). Thus, the query execution on older and newer data reveals additional information.■

To prevent such attacks, we need to appropriately modify our query execution methods. In our technique, we will assume that bins across all rounds are of a fixed size, $|b|$,[6] and the number of tuples for a given attribute value (*i.e.*, location) fits within a bin (*i.e.*, $\leq |b|$). Our idea is inspired by Path-ORAM [36], while we overcome the limitation of Path-ORAM that achieves indistinguishability for query execution by keeping a meta-index structure at the trust entity. Note that Path-ORAM builds a binary tree index on the records. To retrieve a single record, Path-ORAM fetches $O(\log n)$ records and rewrites them under a different encryption. Since Path-ORAM uses an external data structure, it cannot be used for our purpose as argued in §1. Below, we provide our modified query execution strategy.

Executing queries. Let rd_i, rd_j, rd_k, and rd_l be four consecutive rounds of data insertion. Let q be a query that spans over rd_j, rd_k, and rd_l rounds; however, only rounds rd_j and rd_l have bins that satisfy query q. For answering q, the modified query execution method takes the following three steps: (*i*) The enclave fetches the desired bin from rd_j and rd_l rounds by following methods given in §4.2 and §5, with randomly selected $\log |Bin| - 1$ additional bins from each rd_j and rd_l round, where $|Bin|$ are created for each round using Algorithm 2. (*ii*) The enclave fetches $\log |Bin|$ bins from round rd_k, to hide the fact that rd_k does not satisfy the query. (*iii*) For round rd_x, $x \in \{j, k, l\}$, the enclave, first, permutes the retrieved data of rd_x and encrypts with a new key.[7] The newly encrypted data replaces the older data of rd_x.

Aside. We rewrite tuples of fetched bins, when asking a query for another value belonging to the previously fetched bin (*e.g.*, query for l_2 in Example 6.1). The adversary cannot link bins of different rounds of data insertion based on attribute values in bins.

7 SECURITY PROPERTIES

This section presents the desired security requirements, discusses which requirements are satisfied by Concealer, and information leakages from the algorithms. To develop applications on top of spatial time-series dataset at an untrusted $\mathcal{SP}$, a system needs to satisfy the following security properties:

Ciphertext indistinguishability: property requires that any two or more occurrences of a cleartext value look different in the ciphertext. Thus, by observing the ciphertext, an adversary cannot learn anything about encrypted data. Concealer satisfies this property by producing unique ciphertext for each tuple using Algorithm 1 (Line 7).

Data integrity: property requires that if the adversary injects any false data into the real dataset, it must be detected by a trusted entity. Concealer ensures integrity property by maintaining hash chains over the encrypted tuples and sharing encrypted verifiable tags, which helps SGX to detect any inconsistency between the actual data shared by $\mathcal{DP}$ and the data SGX accesses from the disk at $\mathcal{SP}$.

Query execution security: requires satisfying output-size prevention, indistinguishability under chosen keyword attacks (IND-CKA), and forward privacy.

Output-size prevention: property requires that the number of tuples corresponding to a value, *e.g.*, $\mathcal{L}$, (or a value corresponding to a combination of attributes, *e.g.*, $\mathcal{L}$ and $\mathcal{T}$) *i.e.*, the volume of the value, is not revealed, and only the maximum output-size/volume of the value is revealed. Concealer ensures this property by retrieving a fixed-size bin from DBMS into SGX, regardless of the query predicates.

IND-CKA. IND-CKA [11] prevents leakages other than what an adversary can gain through information about (*i*) *metadata*, *i.e.*, size of database/index, known as *setup leakage* $\mathfrak{L}_s$ in [11], and (*ii*) *query execution* that results in *query leakage* $\mathfrak{L}_q$ in [11] and includes search-patterns and access-patterns. Again, note that by revealing the access-patterns, IND-CKA is prone to attacks based on the output-size.

While Concealer leaks $\mathfrak{L}_s$ by the size of database/index and $\mathfrak{L}_q$ by fetching data in the form of a bin, it does not reveal information based on the output-size, except a constant output-size for all query predicates. (Also, since by fetching a bin, it does not reveal which rows of the bin satisfy the answer, it hides partial access-patterns.) Thus, Concealer improves the security guarantees of IND-CKA.

Forward privacy: property requires that newly inserted tuples cannot be linked to previous search queries, *i.e.*, the adversary that have collected trapdoors for previous queries, cannot use them to match newly added tuples. Concealer ensures forward privacy by, first, producing a different ciphertext of an identical value over two different epochs using two different keys (as mentioned in §3), and then, re-encrypting the tuples of different epochs using different keys during query execution spanning over multiple epochs (as mentioned in §6).

Now, we discuss information leakages from different the above-mentioned query execution methods.

BPB information leakage discussion. BPB method prevents the attacks based on output-size by fetching an identical number of tuples for answering any query. It reveals the dataset and index sizes stored in DBMS (as $\mathfrak{L}_s$ condition of IND-CKA [11]). BPB method, also, reveals *partial* access- and search-patterns, which means that for a group of queries it reveals a fixed bin of tuples, and thus, hides which of the tuples of bin satisfy a particular query ($\mathfrak{L}_q$ conditions of IND-CKA). Recall that an index, *e.g.*, B-tree index, on the desired attribute is created by the underlying DBMS. To show that the index will not lead to additional leakages other than $\mathfrak{L}_s$ and $\mathfrak{L}_q$, we follow the identical strategy to prove a technique is IND-CKA secure or not. In short, we need to show that a simulator not having the original data can also produce the

[6] We are not interested in hiding different numbers of tuples in different rounds, but using fake tuples it can be prevented, if desired.

[7] The key k for re-encryption is generated as: $k \leftarrow s_k \| eid \| counter$, where SGX maintains a counter for each round, and increments it by one whenever the data of a round is read in SGX and rewritten.

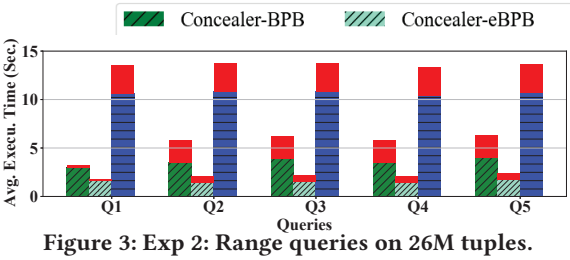


Figure 3: Exp 2: Range queries on 26M tuples.

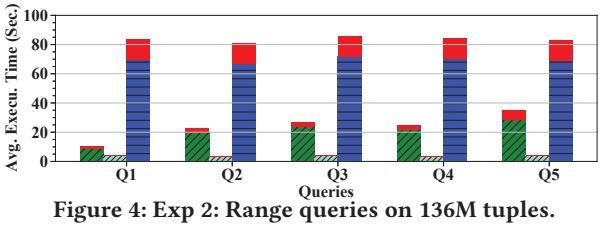


Figure 4: Exp 2: Range queries on 136M tuples.

index attribute based on $\mathfrak{L} = \{\mathfrak{L}_s, \mathfrak{L}_q\}$, *i.e.*, BPB method is secure if a "fake" attribute can mimic the real index attribute, (and hence, mimic the real index). Note that like SSEs, the simulator having only $\mathfrak{L}$ can generate a fake dataset, and hence, the index attribute can mimic the real index attribute; thus, the adversary cannot deduce additional information based on $\mathfrak{L}$.

Also, note that in oblivious Step 3, the enclave generates the same number of real/fake queries regardless of a bin and sorts them using a data-independent algorithm, which hides access-patterns in SGX. Also, it processes all retrieved tuples against the query and does oblivious sorting in Step 4. Thus, it also does not reveal access-patterns (by missing any tuple to process).

eBPB and winSecRange information leakage discussion. eBPB method incurs leakages $\mathfrak{L}_s$ and $\mathfrak{L}_q$. Based on $\mathfrak{L}_q$, we may reveal that a range query is spanning over at most ℓ cells. Hence, *it may also reveal the exact data distribution*, by fetching the same real tuple multiple times for multiple range queries, which we illustrated in Example 5.1.2. To overcome such information leakage, winSecRange fetches a fixed size interval, regardless of the query range. Thus, while winSecRange reveals $\mathfrak{L}_s$ and $\mathfrak{L}_q$, it does not reveal any information based on the output-size.

Insert operation information leakage discussion. Our insert operation satisfies forward privacy property. Since for encrypting tuples of an epoch, we generate a key that is unique among all keys generated for any epoch. Thus, based on the previous query trapdoors, the adversary cannot use them to link new tuples. Furthermore, our insert operation hides the distribution leakage due to executing queries over multiple epochs, since we fetch additional tuples from each epoch that lies in the query range and re-write all tuples.

8 EXPERIMENTAL EVALUATION

This section shows the experimental results of Concealer under various settings and compares them against prior cryptographic approaches.

8.1 Datasets, Queries, and Setup

Dataset 1: Spatial time-series data generation. To get a real spatial time-series dataset, we took our organization WiFi user connectivity dataset over 202 days having 136M(illion) rows. The IT department manages more than 2000 WiFi access-points (AP) by which they collect tuples of the form $\langle l_i, t_i, o_i \rangle$ on which they implemented Algorithm 1 prior to sending WiFi data to us. In this data, each of 2000+ APs is considered as a location. We created two types of WiFi datasets: (*i*) a small dataset of 26M WiFi connectivity rows collected over 44 days, and (*ii*) a large dataset of 136M rows (of 14GB) collected over 202 days. For Concealer Algorithm 1, which produces encrypted data as shown in Table 2c, we fixed a grid of $490 \times 16,000$ and allocated 87,000 cell-ids that resulted in two vectors *cell_id*[] and *c_tuple*[] of size 31MB. Data was encrypted using AES-256. This dataset has also skewed over the number of tuples at locations in a given time. For example, the minimum number of rows at all locations in an hour was ≈6,000, while the maximum number of rows at all locations in an hour was ≈50,000.

Dataset 2: TPC-H dataset. Since WiFi dataset has only three columns, to evaluate Concealer's practicality in other types of data with more columns, we used 136M rows of LineItem table of TPC-H benchmark. We selected nine columns (Orderkey (OK), Partkey (PK), Suppkey(SK), Linenumber (LN), Quantity, Extendedprice, Discount, Tax, Returnflag). This dataset contains large domain values, also; *e.g.*, in OK column, the domain value varies for 1 to 34M. We created (*i*) two indexes on attributes ⟨OK, LN⟩ and ⟨OK, PK, SK, LN⟩, (*ii*) two filters on concatenated values of ⟨OK, LN⟩ and ⟨OK, PK, SK, LN⟩, and (*iii*) one value that is the encryption of the concatenated values of all remaining five attributes. We used a $112,000 \times 7$ grid for index ⟨OK, LN⟩ and a $1500 \times 100 \times 10 \times 7$ grid for index ⟨OK, PK, SK, LN⟩. Each grid was allocated 87,000 cell-ids. The size of *cell_id*[] and *c_tuple*[] vectors for both grids was 54MB. Data is encrypted using Algorithm 1 and AES-256.

Queries. Table 4 lists sample queries supported by Concealer on spatial time-series data. These queries as mentioned in §2.1 provide aggregate (Q1-Q3) and individualized (Q4-Q5) applications. On TPH-C data, we executed count, sum, min/max queries.

Setup. The IT department (worked as $\mathcal{DP}$) had a machine of 16GB RAM. Our side (worked as $\mathcal{SP}$) had a 16GB RAM Intel Xeon E3 machine with Intel SGX. At $\mathcal{SP}$, MySQL is used to store data, and ≈8,000 lines of code in C is written for query execution.

We evaluate both versions of Concealer depending on the security of SGX: (*i*) one that assumes SGX to be completely secure against side-channel attacks, denoted byConcealer, and (*ii*) another that assumes SGX is not secure against side-channel attack (cache-line, branch shadow, page-fault attacks) and hence performs the oblivious computation in SGX (given in §4.3). denoted by Concealer+. In all our experiments, we show the overhead of preventing the side-channel attacks using red color.

8.2 Concealer Evaluation

This section evaluates Concealer on different aspects such as scalability, dynamic data insertion, the impact of the range length, and the number of cell-ids.

Exp 1: Throughput. Since Concealer is designed to deal with data collected during an epoch arriving continuously over time, we measured the throughput (rows/minute) that Concealer can sustain to evaluate its overhead at the ingestion time. Algorithm 1 can encrypt ≈37,185 WiFi connectivity tuple per minute. Also, it sustains our organization-level workload on the relatively weaker machine used for hosting Concealer.

Exp 2: Scalability of Concealer. To evaluate the scalability of Concealer, we compare the five queries as specified in Table 4 on the two WiFi datasets.

Point query. For point query, we executed a variant of Q1 when the time is fixed to be a point (instead of a range). Table 5 shows the average time taken by 5 randomly selected point queries (each executed 10 times). Note that, in Concealer, since the time taken by point queries is dependent upon the number of tuples allocated to the same cell-id (*i.e.*, the bin size) that was 2,378 rows

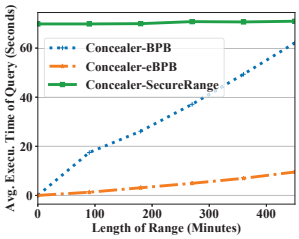


Figure 5: Exp 3: Range length impact.

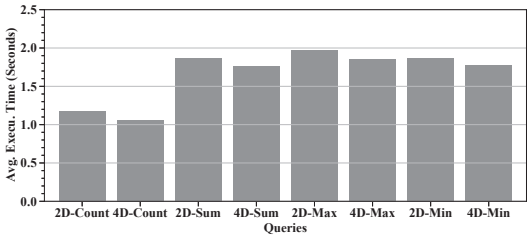


Figure 6: Exp 5: Query performance on TPC-H data.

from small and 6,095 rows from large datasets. Table 5 shows that Concealer with secure SGX using BPB method took 0.23s on small and 0.90s on large datasets, while Concealer+ with the current non-secure SGX using BPB method took 0.37s on small and 1.38s on large datasets. ***Time in Concealer+ increases compared to Concealer***, since we need to obliviously form the queries and obliviously filter the tuples in Concealer+, (and that implements a data-independent sorting algorithm; see §4.3). Here, executing the same query on cleartext data took 0.03s on small and 0.05s on large datasets.

	Small dataset (26M)	Large dataset (136M)
Cleartext processing	0.03s	0.05s
Concealer (secure SGX)	0.23s	0.90s
Concealer+ (non-secure SGX)	0.37s	1.38s

Table 5: Exp 2: Scalability of point query.

Range queries. To evaluate range queries, we set the default time range for queries Q1-Q5 specified in Table 4 to 20min (Exp 4 will study the impact of different range lengths). Figures 3 and 4 show the results as an average over 5 queries (each executed 10 times). We compare BPB, eBPB (§5.1), and winSecRange (§5.2) with both Concealer and Concealer+.

Recall that BPB method answers a range query by converting it into many point queries and fetches bins corresponding to each point query; while eBPB method fetches rows corresponding to top-ℓ cells, which cover the given range. In Concealer, a cell covers ≈18min. Thus, for a range of 20min, BPB and eBPB methods fetch at most 3 bins and at most 3 cells, respectively, for query Q1. Thus, for answering Q1, BPB fetches ≈6K rows from small and ≈18K rows from large datasets, and eBPB fetches ≈1.5K rows from small and ≈3K rows from the large dataset. Since eBPB retrieves few numbers of rows compared to BPB, in sGX, eBPB performs better than BPB (see Figures 3 and 4). Concealer+ again takes more time compared to Concealer for both eBPB and BPB, due to oblivious operations. Note that in queries Q2-Q5, we use more locations; thus, the number of rows retrieved changes accordingly, and hence, the processing time also changes, as shown in Figures 3 and 4.

For winSecRange, we set the range length on the time attribute to 8 hours in case of small and ≈1-day in case of large datasets. Thus, by fetching data for 1-day in the case of the large dataset, the enclave can execute any range query that is of a smaller time length. As expected, winSecRange took more time to execute range queries on both datasets, since it fetches and processes more rows (≈70K rows from small and ≈400K from large datsets). While it takes more time compared to BPB and eBPB, it prevents the attack by sliding the time window (as shown in Example 5.1.2), thereby, prevents revealing output-size attacks due to the sliding time window. Further, winSecRange under Concealer+ took more time compared to winSecRange under Concealer, due to oblivious operations. Recall that under Concealer, SGX architecture is vulnerable to side-channel attacks.

Exp 3. Impact of range length. Figure 5 shows the impact of the length in a range query on Concealer, by executing Q1 (see Table 4) with different time lengths over the large dataset and compares three approaches BPB, eBPB, and winSecRange. In Concealer, a cell covers ≈18min. Thus, for instance, for a range of 100min, BPB and eBPB methods fetch at most 7 bins and at most 7 cells, respectively. As expected, as the length of range increases, the number of rows to be fetched from the DBMS also increases, thereby, the processing time at secure hardware increases. As mentioned in Exp 2, for the large dataset, the range length is set to ≈1-day for winSecRange method; hence, fetching/processing more tuples takes more time and remains almost constant for the given length of queries.

Exp 4. Impact of dynamic insertion. We also investigated how does Concealer support dynamic insertion of WiFi dataset. We initiated Algorithm 1 for an hour of WiFi data at the peak hour, which included ≈50K tuples. For each insertion round, the grid size was $20 \times 1,250$ with 400 allocated cell-ids, and vectors $cell\_id[]$ and $c\_tuple[]$ of size ≈100KB were generated. In non-peak hours, we received at least ≈6K real rows. Recall that we are not interested in hiding peak vs non-peak hour data. Thus, all rows of each hour were sent using Algorithm 1. The query execution performance on dynamically inserted data depends on the number of rounds over which a query spans. For each round, we need to load the two vectors and fetch $\log |Bin|$ bins, as described in §6. For peak hour data, we obtained 146 bins storing ≈400 tuples, in each, using BPB method (§4.2) that resulted in ≈3K row retrieval. On this data, it took at most ≈4s to execute a query, re-encrypting tuples, and writing them, for Concealer.

Exp 5. Concealer on TPC-H data. To evaluate Concealer's practicality in other types of queries, we executed two-dimensional (2D) and four-dimensional (4D) count, sum, maximum, and minimum queries on LineItem table using Concealer. 2D (and 4D) queries involved OK and LN (and OK, PK, SK and LN) attributes. Similar to the point query execution on WiFi dataset, 2D and 4D queries on TPC-H data require to fetch tuples allocated the same cell-id (in the same bin) according to Algorithm 2. Thus, the query execution performance is dependent on the bin size, which was 400 rows for 2D grid and 6,258 for 4D grid. The query execution results are shown in Figure 6.

Figure 6 shows that Concealer using BPB method took ≈1s to 2s on TPC-H dataset. Also, observe that the performance of count queries is ≈36%–40% better than the others queries, since count queries do not need to decrypt retrieved rows and it executed string matching on the filter column to produce the answer. In contrast, other queries that require exact values of the attributes decrypt retrieved rows, and hence, incur the overhead.

8.3 Other Cryptographic Techniques

Since in our setting $\mathcal{SP}$ uses secure hardware, we need to compare Concealer against a system that supports database operations using SGX. Thus, we selected an open-source SGX-based system: Opaque [42].

Comparison between Opaque and Concealer. Opaque supports mechanisms to execute databases queries on encrypted data by first reading the entire data in the enclave, decrypting them, and then providing the answer. Note that both Opaque

System	Q1	Q2	Q3	Q4	Q5
Opaque	>10 m	>10 m	>10 m	>10 m	>10 m
Concealer eBPB	3.6 s	2.8 s	3.4 s	3 s	4s
Concealer winSecRange	70 s	67.2 s	71.9 s	70.2 s	68.9 s

Table 6: Exp 7: Range queries: Opaque vs Concealer.

and Concealer assume that SGX is secure against side-channel attacks, and hence, both reveal access-patterns. Thus, this is a fair comparison of the two systems, while Concealer avoids reading the entire dataset due to using the index and pushing down the selection predicate. Under this comparison, we execute point and range queries using Opaque and Concealer.

Further, note that since Concealer+ completely hides access-patterns inside SGX and partially hides access-patterns when fetching data in the form of fixed-size bins from the disk, we do not directly compare Concealer+ and Opaque due to different level of security offered by two systems.

Exp 6: Point queries on WiFi data. Opaque took more than 10min on both WiFi datasets for executing a variant of Q1 when the time is fixed to be a point, since *Opaque requires reading the entire dataset*. For the same query, Concealer took at most 0.23s on 26M and 0.9s on 136M rows.

Further, to execute the same query, Concealer+ took ≈1.4s. Thus, it shows that Concealer+, which hides access-patterns inside the enclave and prevents the output-size attack, is significantly better than Opaque.

Exp 7: Range queries on WiFi data. In all range queries Q1-Q5 on WiFi data, Concealer's eBPB and winSecRange algorithms take at most 4s and 71.9s over the large dataset compared to Opaque that took at least 10min in any query; see Table 6.

Further, to execute the same queries, Concealer+ takes at most 90s over the large dataset, which shows better performance of Concealer+ than Opaque in the case of range queries also.

Note. Except for Opaque, we did not experimentally compare Concealer against cryptographic techniques, since such techniques either offer different security levels [12, 22, 23], or do not scale to large data (*e.g.*, [4, 15]) for which we have designed Concealer, or are not publicly available. Thus, we decide to compare Concealer results with the reported result in different papers. Previous works on secure OLAP queries either support limited operations, reveal data due to DET or OPE, or scale to a smaller dataset. For example, Monomi [38], Seabed [29], [14], and [24] reveal data due to DET or OPE. Nevertheless, Seabed supports a huge dataset (≈1.75B rows). Novel SSEs, *e.g.*, [12, 22, 23], are very efficient, as given in their experiments; however, over 5M rows and susceptible to output-size attacks. We also checked access-pattern-hiding cryptographic work (*e.g.*, DSSE [15] and Jana [4]) that are prone to output-size attacks; however, as expected, these systems are slow due to using highly secure cryptographic techniques that incur overheads and/or do not support a large dataset. E.g., an industrial MPC-based system Jana took 9 hours to insert 1M LineItem rows, while executing a simple query took 532s.

9 CONCLUSION

This paper proposed Concealer that blends a carefully chosen encryption method with mechanisms to add fake tuples and exploits secure hardware to efficiently answer OLAP-style queries. We applied Concealer to real spatial time-series datasets, as well as, synthetic TPC-H data, and demonstrated scalability to large-sized data. Since Concealer allows indexing, its performance is similar to SSEs.Concealer offers two key advantages over existing SSEs: first, it does not require new data structures to incorporate into databases and leverages existing index structures of modern databases. Second (and perhaps more importantly), Concealer offers a higher level of security, in addition to being IND-CKA, which existing SSEs are, by preventing leakage of data distributions via output-size. *Due to space restrictions, we omit the query workload handling method, proof of Theorem 4.1, and experiments related to the overhead of verification, impact of the number of cells, and impact of different bin-sizes from this paper, and will be given in the full version (https://arxiv.org/abs/2102.05238).*

REFERENCES

[1] Blyncsy Inc. https://blyncsy.com/mercury-contact-tracing.
[2] R. Agrawal et al. Order-preserving encryption for numeric data. In *SIGMOD*, pages 563–574, 2004.
[3] P. Antonopoulos et al. Azure SQL database always encrypted. In *SIGMOD*, pages 1511–1525, 2020.
[4] D. W. Archer et al. From keys to databases - real-world applications of secure multi-party computation. *Comput. J.*, 61(12):1749–1771, 2018.
[5] K. E. Batcher. Sorting networks and their applications. In *AFIPS*, 1968.
[6] J. Bater et al. SMCQL: secure query processing for private data networks. *Proc. VLDB Endow.*, 10(6):673–684, 2017.
[7] D. Cash et al. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679, 2015.
[8] E. G. Coffman, Jr. et al. Approximation algorithms for NP-hard problems. chapter Approximation algorithms for bin packing: a survey. 1997.
[9] V. Costan et al. Intel SGX explained. *IACR Cryptology ePrint*, 2016:86, 2016.
[10] V. Costan et al. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX*, pages 857–874, 2016.
[11] R. Curtmola et al. Searchable symmetric encryption: Improved definitions and efficient constructions. *JCS*, 19:895–934, 2011.
[12] I. Demertzis et al. Practical private range search revisited. In *SIGMOD*, pages 185–198, 2016.
[13] A. Dhar et al. Proximitee: Hardened SGX attestation by proximity verification. In *CODASPY*, pages 5–16, 2020.
[14] T. Ge et al. Answering aggregation queries in a secure system model. In *VLDB*, pages 519–530, 2007.
[15] Y. Ishai et al. Private large-scale databases with distributed searchable symmetric encryption. In *CT-RSA*, pages 90–107, 2016.
[16] M. S. Islam et al. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.
[17] S. Jarecki et al. Updatable oblivious key management for storage systems. In *CCS*, pages 379–393, 2019.
[18] S. Kamara et al. Computationally volume-hiding structured encryption. In *EUROCRYPT*, pages 183–213, 2019.
[19] G. Kellaris et al. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.
[20] S. Lee et al. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX*, pages 557–574, 2017.
[21] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Trans. on Computers*, 100(4):344–354, 1985.
[22] R. Li et al. Fast range query processing with strong privacy protection for cloud computing. *PVLDB*, 7(14):1953–1964, 2014.
[23] R. Li et al. Adaptively secure conjunctive query processing over encrypted data for cloud computing. In *ICDE*, pages 697–708, 2017.
[24] C. C. Lopes et al. Processing OLAP queries over an encrypted data warehouse stored in the cloud. In *DaWaK*, pages 195–207, 2014.
[25] P. Mishra et al. Oblix: An efficient oblivious search index. In *IEEE SP*, pages 279–296, 2018.
[26] M. Naveed. The fallacy of composition of oblivious RAM and searchable encryption. *IACR Cryptology ePrint Archive*, 2015.
[27] M. Naveed et al. Inference attacks on property-preserving encrypted databases. In *CCS*, pages 644–655, 2015.
[28] O. Ohrimenko et al. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, pages 619–636, 2016.
[29] A. Papadimitriou et al. Big data analytics over encrypted datasets with seabed. In *OSDI*, pages 587–602, 2016.
[30] V. Pappas et al. Blind seer: A scalable private DBMS. In *IEEE SP*, 2014.
[31] S. Patel et al. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *CCS*, pages 79–93, 2019.
[32] R. Poddar et al. Arx: An encrypted database using semantically secure encryption. *Proc. VLDB Endow.*, 12(11):1664–1678, 2019.
[33] C. Priebe et al. Enclavedb: A secure database using SGX. In *IEEE SP*, pages 264–278, 2018.
[34] M. Shih et al. T-SGX: eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.
[35] D. X. Song et al. Practical techniques for searches on encrypted data. In *IEEE SP*, pages 44–55, 2000.
[36] E. Stefanov et al. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM*, 65(4):18:1–18:26, 2018.
[37] A. Trivedi and other. WiFiTrace: Network-based contact tracing for infectious diseases using passive WiFi sensing. *CoRR*, abs/2005.12045, 2020.
[38] S. Tu et al. Processing analytical queries over encrypted data. *PVLDB*, pages 289–300, 2013.
[39] J. Wang et al. Interface-based side channel attack against Intel SGX. *CoRR*, abs/1811.05378, 2018.
[40] W. Wang et al. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, pages 2421–2434, 2017.
[41] M. Xu et al. Hermetic: Privacy-preserving distributed analytics without (most) side channels.
[42] W. Zheng et al. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, pages 283–298, 2017.

Evaluation of Hardening Techniques for Privacy-Preserving Record Linkage

Martin Franke
University of Leipzig
Germany
franke@informatik.uni-leipzig.de

Ziad Sehili
University of Leipzig
Germany
sehili@informatik.uni-leipzig.de

Florens Rohde
University of Leipzig
Germany
rohde@informatik.uni-leipzig.de

Erhard Rahm
University of Leipzig
Germany
rahm@informatik.uni-leipzig.de

ABSTRACT

Privacy-preserving record linkage aims at integrating person-related data from different sources while protecting the privacy of individuals by securely encoding and matching quasi-identifying attributes, like names. For this purpose Bloom-filter-based encodings have been frequently used in both research and practical applications. Simultaneously, however, weaknesses and attack scenarios were identified emphasizing that Bloom filters are in principal susceptible to cryptanalysis. To counteract such attacks, various encoding variants and tweaks, also known as hardening techniques, have been proposed. Usually, these techniques bear a trade-off between privacy (security) and the linkage quality outcome. Currently, a comprehensive evaluation of the suggested hardening methods is not available. In this work, we will therefore review and categorize available Bloom-filter-based encoding schemes and hardening techniques. We also comprehensively evaluate the approaches in terms of privacy (security) and linkage quality to assess their practicability and their effectiveness in counteracting attacks.

1 INTRODUCTION

Linking records from different independent sources is an essential task in research, administration and business to facilitate advanced data analysis [6]. In many applications, these records are about individuals and thus contain sensitive information, e. g., personal, health, criminal or financial information. Due to several laws and regulations, data holders have to protect the privacy of individuals [33]. As a consequence, data holders have to ensure that no sensitive or confidential information is revealed during a linkage process.

Privacy-preserving record linkage (PPRL) addresses this problem by providing techniques for linking records referring to the same real-world entity while protecting the privacy of these entities. In contrast to traditional record linkage [6], PPRL encodes sensitive identifying attributes, also known as quasi-identifiers, for instance, names, date of birth or addresses, and then conduct the linkage on the encoded attribute values.

Over the last years, numerous PPRL approaches have been published [33]. However, many approaches are not suited for real-world applications as they either are not able to sufficiently handle dirty data, i. e., erroneous, outdated or missing values, or do not scale to larger datasets. More recent work mainly focuses on encoding techniques utilizing Bloom filters [2] as error-tolerant and privacy-preserving method to encode records containing sensitive information. While Bloom-filter-based encodings have become the quasi-standard in PPRL approaches, several studies analyzed weaknesses and implemented successful attacks on Bloom filters [7, 8, 18, 20, 21, 24]. In general, it was observed that Bloom filters carry a non-negligible re-identification risk because they are vulnerable to frequency-based cryptanalysis. In order to prevent such attacks, various Bloom filter hardening techniques were proposed [7, 25]. Such techniques aim at reducing patterns and frequency information that can be obtained by analyzing the frequency of individual Bloom filters or (co-occurring) 1-bits.

Previous studies on Bloom filter hardening techniques only consider individual methods and do not analyze the effects of combining different approaches. Moreover, many of the proposed hardening techniques have received only limited evaluation on small synthetic datasets making it hard to assess the possible effects on the linkage quality.

The aim of this work is to review hardening techniques proposed in the literature and to evaluate their effectiveness in terms of achieving high privacy (security) and linkage quality.

In particular, we make the following contributions:

- We survey Bloom filter variants and hardening techniques that have been proposed for use in PPRL scenarios to allow secure encoding and matching of sensitive person-related data.
- We categorize existing hardening techniques to generalize the Bloom filter encoding process and thus highlight the different possibilities for building tailored Bloom filter encodings that meet the privacy requirements of individual application scenarios.
- We explore additional variants of hardening techniques, in particular salting utilizing blocking approaches and attribute-specific salting on groups of attributes.
- We propose and analyze measures that allow us to quantify the privacy properties of different Bloom filter variants.
- We comprehensively evaluate different Bloom filter variants and hardening techniques in terms of privacy (security) and linkage quality using two real-world datasets containing typical errors and inconsistencies.

2 BLOOM FILTER

The use of Bloom filters [2] for PPRL has been proposed by Schnell and colleagues [26] and has become the quasi-standard for recent PPRL approaches in both research and real applications [33].

2.1 Basics

A Bloom filter (BF) is a space-efficient probabilistic data structure for representing a set $E = \{e_1, \ldots, e_n\}$ of n elements or features and testing set membership. Therefore, a bit vector of fixed size m is allocated and initially all bits are set to zero. A set of $\mathbf{k}$ hash functions is selected where each function $H_1, \ldots, H_k$ outputs a value in $[0, m-1]$. To represent the set E in the BF, each element is (hash) mapped to the BF by using each of the k hash functions and setting the bits at the resulting positions to one.

To check the membership of an element, the same hash functions are calculated and the bits at the resulting positions are checked. If all bits are set to one, the element probably is in the set. Due to collision, i. e., two or more elements may set the same bit position for the same or different hash functions, BFs have a false-positive probability that is $\text{fpr} = (1 - e^{\frac{k \cdot n}{m}})^k$ [3]. On the other hand, if at least one bit is zero, the element is definitively not in the set.

Union and intersection of BFs with the same size and set of hash functions can be implemented with bit-wise OR and AND operations respectively. While the union operation is lossless, i. e., the resulting BF will be equal to a BF that was build using the union of the two sets, the intersection operation produces a BF that may have a larger false-positive probability [3].

By using BF union and intersection, set-based similarity measures can be used to calculate the similarity of two BFs. Here the **Jaccard coefficient** is frequently used as a similarity measure. Given two BFs x, y the Jaccard coefficient is defined as

$$J(x, y) = \frac{|x \cap y|}{|x \cup y|} = \frac{|x \text{ AND } y|}{|x \text{ OR } y|}$$

For instance, given the BFs $x = [\mathbf{1}0011001]$ and $y = [\mathbf{0}0011001]$ the Jaccard coefficient is $3/4$. The BF similarity is an approximation of the similarity of the underlying (represented) sets.

2.2 Utilization in PPRL

The main idea for utilizing BFs in PPRL scenarios is to use a BF to represent the records attribute values, i. e., all quasi-identifying attributes of a person that are relevant for linkage, e. g., first name, last name and date of birth. The BFs hash functions need to be cryptographic (one-way) hash functions that are keyed (seeded) with a secret key $\mathcal{S}$, i. e., keyed-hash message authentication codes (HMACs) like MD5 or SHA-1 [23]. For approximate matching, the granularity of the record attributes is increased by segmentation into features. A widely used approach is to split the attribute values into small substrings of length q, called **q-grams**, typically setting $1 \leq q \leq 4$. Consequently, in PPRL a BF represents a set of attribute value segments, that we term record (attribute) features. Thus, the number of common 1-bits of two BFs approximates the number of common (overlapping) features between two records.

2.2.1 Types. There are two ways of encoding records into BFs: either one BF is built for each record attribute, which is known as field- or **attribute-level BF**, or a single BF is built for all relevant attributes, which is known as **record-level BF**. For constructing record-level BFs there are two approaches: The first approach [27] builds a single BF in which all record attributes are hashed. The second approach [9] first constructs field-level BFs and then selects bits from these individual BFs according to the weight of the respective attribute. In this work, we will focus on the first approach since it is heavily used in literature and practice [33]. We illustrate the basic BF building process in Fig. 1. By using

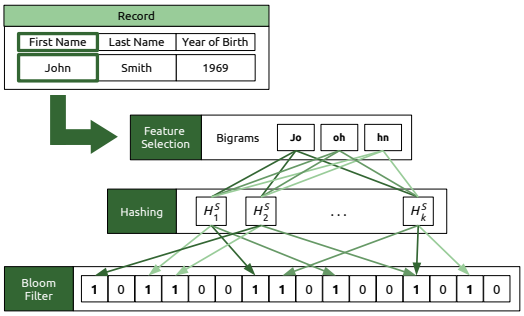


Figure 1: Basic Bloom filter building process.

the first name attribute the figure shows how an attribute value (*'John'*) is segmented into q-gram segments (here $q = 2$), which are then mapped to the bit vector using the k hash functions. Other attributes are mapped in the same way, although a different segmentation strategy can be used. The advantage of using field-level BFs is that individual BFs are produced allowing the use of sophisticated matching techniques known from traditional record linkage, e. g., classification based on attribute weights and attribute error rates, as well as approaches for handling composite fields, for instance, name attributes with compounds (multiple given names). However, as we discuss in the next section, field-level BFs fulfill much weaker privacy properties compared to record-level BFs.

2.2.2 Privacy Properties. The privacy-preserving property of BFs rely on the following aspects:

(1) An adversary has no information on how the record features are obtained, e. g., selected attributes or length of substrings (q-grams).
(2) The selected hash functions, the secret key $\mathcal{S}$ and thus the hash mapping of record features to bit positions is unknown to an adversary. In particular, the use of keyed hash functions is essential to prevent dictionary attacks.
(3) Due to collisions multiple record features will map to a single bit position in general. Keeping the BF size m fixed, the more hash functions are used and the more features are mapped to the BF, the higher will be the number of collisions and thus the confusion.
(4) There is no coherence or positional information: Since a BF encodes a *set* of record features, it is not obvious from where features were obtained, i. e., within an attribute and for record-level BF even from which attribute.

However, BFs are susceptible to frequency attacks as the frequencies of set bit positions correspond to the frequencies of record features. Thus, frequently (co-)occurring record features will lead to frequently set bit positions or even to frequent BFs in the case of field-level BFs. By using publicly available datasets containing person-related data, e. g., telephone books, voter registration databases, social media profiles or databases about persons of interests like authors, actors or politicians, an adversary can estimate the frequencies of record features and then try to align those frequencies to the BFs bit frequencies.

A successful re-identification of attribute values encoded in BF is a real threat as shown by several attacks proposed in the literature. Earlier attacks, namely [18, 20, 21, 24], often exploit the hashing method used in [27], the double-hashing scheme, that combines two hash functions to implement the k BF hash

Table 1: Overview of surveyed Bloom filter hardening techniques.

Subject of modification	Technique	Reference	Description
Bloom filter input	Avoidance of padding	[24, 25]	No use of padded q-grams as BF input due to their higher frequency.
	Standardization of attribute lengths	[24, 25]	The length of attribute values is unified to avoid exceptionally short or long values.
Hashing mechanism	Increasing the number of hash functions (k)	[26, 27]	Using more hash functions (k) while keeping the Bloom filter size (m) fixed will lead to more collisions and thus a higher number of features that are mapped to each position.
	Random hashing	[24]	Replacement for the double-hashing scheme [26] which can be exploited in attacks [24].
	Attribute weighting	[9, 32]	Record features are hashed with a different number of hash functions (k) depending on the weight of the attribute from which they were obtained.
	Salting	[24, 27]	Record features are hashed together with an additional attribute specific and/or record specific value.
Output Bloom filter	Balancing	[28]	Each Bloom filter is concatenated with a negative copy of itself and then the underlying bits are permuted.
	XOR-folding	[29]	Each Bloom filter is split into halves which are then combined using the bit-wise XOR-operation.
	Re-hashing	[25]	Sliding window approach where the Bloom filter bits in each window are used to generate a new set of bits.
	Rule90	[30]	Each Bloom filter bit is replaced by the result of XOR-ing its two neighbouring bits.
	Random noise	[1, 24–26, 28]	Bloom filter bits are changed randomly.
	Fake injections	[16]	Addition of artificial records and thus Bloom filters.

functions. This hashing method can easily be replaced by using independent hash functions or other techniques as discussed in Sec. 3.2.1. Furthermore, these attacks rely on many unrealistic assumptions, for instance, that the encoded records are a random sample of a resource known to the adversary [20, 24] or that all parameters of the BF process, including used secret keys for the hash functions, are known to the adversary [21]. However, recent frequency-based cryptanalysis attacks, namely [7] and in particular [8], are able to correctly re-identify attribute values without relying on such assumptions. These attacks are the more successful, the fewer attributes are encoded in a BF and the larger the number of encoded records. Overall, the attacks show the risk of re-identification when using BFs, especially field-level BFs. In this work, we will focus only on record-level BFs.

3 BLOOM FILTER VARIANTS AND HARDENING METHODS

In the following, we review different variations within the BF encoding process. In general, these variations will affect both the BFs privacy and similarity-preserving (matching) properties. Approaches that try to achieve a more uniform frequency distribution of individual BFs or set bit positions are also known as hardening techniques as they are intended to make BF encodings more robust against cryptanalysis. An overview of these techniques is given in Tab. 1. We divide the approaches into three categories: (A) approaches that alter the way of selecting features from the records attributes values, (B) approaches that modify the BFs hashing process and (C) approaches that modify already existing BFs by changing or aggregating bits. In the following subsections, we will describe the approaches of each category.

3.1 Record Feature Selection

We will first focus on how features are selected from the record's attributes. In the encoding process, at first, all attribute values are pre-processed to bring them into the same format and to reduce data quality issues. After that, all linkage-relevant attributes, i. e., the quasi-identifiers of a person, are transformed into their respective feature set. Such features are pieces of information that are usually obtained by segmenting the attribute values into chunks or tokens. This is necessary because instead of a binary decision for equality (true/false), approximate linkage is desired resulting in similarity scores ranging from zero (completely different) to one (equal).

3.1.1 Standardization of Attribute Lengths. Quasi-identifiers, such as names and addresses, show high variation and skewness leading to significant differences in the length of attribute values [11]. For instance, multiple given names, middle names or compound surnames (e. g., *'Hans-Wilhelm Müller-Wohlfahrt'*) will lead to exceptionally long attribute values and consequently a comparatively large amount of 1-bits in the resulting BF. The same applies for very short names (e. g., *'Ed Lee'*) resulting in very few 1-bits in the BF. By analyzing the number of 1-bits in a set of BFs, an adversary can gain information on the length of encoded attribute values. To address this problem, the length of the quasi-identifiers should be standardized by sampling, deletion or stretching of the attribute values [25]. Stretching can be implemented by concatenating short attribute values with (rarely occurring) character sequences.

3.1.2 Segmentation Strategy. The standard segmentation strategy adopted from traditional record linkage is to transform all quasi-identifiers into their respective **q-gram** set. A q-gram set is a set of all consecutive character sequences of length q that can be built from the attribute's string value by using a sliding window approach. For instance, setting $q = 3$ the value *'Smith'* will produce the q-gram set {*Smi*, *mit*, *ith*}. The idea behind building these q-gram sets is that they allow approximate string comparisons by calculating the number of q-grams two sets have in common. To directly obtain a similarity value, any set-based similarity measure, e. g., Jaccard coefficient, can be used.

The choice of q is important since it can affect the linkage quality. Usually, q is selected in the range [1..4] while most approaches setting $q = 2$. In general, larger values for q are more sensitive to single character differences, e. g., the values *'Smith'* and *'Smyth'* will have two bigrams ($q = 2$), i. e., *'Sm'* and *'th'*, but zero trigrams ($q = 3$) in common. However, choosing a larger q also increases to number of possible q-grams, e. g., for $q = 2$ at maximum $26^2 = 676$ while for $q = 3$ at maximum $26^3 = 17\,576$ are possible. Overall, larger values for q tend to be less error-tolerant and thus possibly lead to missing matches. On the other hand, larger q's are more distinctive and thus tend to reduce false-positives.

As can be seen from the example above, for $q > 1$ each character will contribute to multiple q-grams except the first and last character. Thus, a common extension is to construct **padded q-grams** by surrounding each attribute value with $q - 1$ special characters at the beginning and the end. For our example the

padded q-gram set will be {*++S*, *+Sm*, *Smi*, *mit*, *ith*, *th-*, *h--*}. By using padded q-grams strings with the same beginning and end but variations in the middle will reach larger similarity values, while strings with different beginning and end will produce lower similarity values compared to standard q-grams [6]. It is important to note, that padded q-grams are among the most frequent q-grams and thus can ease any frequency alignment attacks.

There are several other extensions for generating q-grams, two of which have been used in traditional record linkage, but so far not for PPRL: **positional q-grams** and **skip-grams** [6]. Positional q-grams add the information from which position the q-gram was obtained. For our running example, the positional q-gram set for $q = 3$ is $\{(Smi, 0), (mit, 1), (ith, 2)\}$. When determining the overlap between two positional q-gram sets, only the q-grams at the same position or within a specific range are considered. Positional q-grams will be more distinctive and thus tend to reduce false-positives and even the frequency distribution. The idea of skip-grams is to not only consider consecutive characters but to skip one or multiple characters. Depending on the defined skip length multiple skip-gram sets can be created and used in addition to the regular q-gram set.

So far, only a few alternatives to q-grams have been investigated. In [17] and [31] the authors explore methods for handling numerical attribute values. Besides, arbitrary substrings of individual length or phonetic codes, such as Soundex [6], are possible approaches that can be used for feature extraction.

3.2 Modification of the Hashing Mechanism

After transforming all quasi-identifiers in their respective feature set, the features of each set are hashed into one record-level BF. As discussed in Sec. 2.2.2, we do not further consider field-level BFs due to their vulnerabilities. For PPRL several modifications of the standard hashing process of BFs have been proposed which we will discuss below.

3.2.1 Hash Functions. As described in Sec. 2.2, by default k independent (cryptographic) hash functions are used in conjunction with a private key $\mathcal{S}$ to prevent dictionary attacks. However, the authors of [27] proposed the usage of the so-called double-hashing scheme. This scheme only uses two independent hash functions G_1, G_2 to implement the BFs k hash functions. Each hash function is then defined as $H_i(x) = (G_1(x) + (i-1) \cdot G_2(x)) \mod m, \forall i \in \{1, \ldots, k\}$. The attacks described in [18, 24] showed that this specific scheme can be successfully exploited. As a consequence, an alternative method, called **random hashing**, was proposed [24] that utilizes a pseudo-random number generator to calculate the hash values. Therefore, the random number generator is seeded with the private key $\mathcal{S}$ and the actual input of the hash function, i. e., a certain record feature. No attacks against this method are known at present.

3.2.2 Salting. Salting is a well-known technique in cryptography that is often used to safeguard passwords in databases [22]. The idea is to use an additional input, called salt, for the hash functions to flatten the frequency distribution. Already in [27] it is mentioned that a different cryptographic secret key $\mathcal{S}_a$ can be used for each record *attribute a*. We term such kind of key as **attribute salt**. By using this approach, the same feature will be mapped to different positions if it originates from different attributes. For instance, given the first name *'thomas'* and the last name *'smith'* the bigram *'th'* will produce different positions. This approach will smoothen the overall frequency distribution and also reduce false-negatives since features from different attributes will not produce common 1-bits (except due to collision). However, the BF's ability to match exchanged attributes, e. g., transposed first and middle name, is lost. If such errors occur repeatedly this will lead to missing matches. As a compromise, we propose to define groups of attributes, where transpositions are expectable. Then, the same key is used for each attribute from the same group. For instance, all name-related attributes (first name, middle name, last name) could form a group.

Another salting variant is proposed in [24], where *for each record* a specific salt is selected and then used as key for the BFs k hash functions. Therefore, we term such keys as **record salt**, since they depend on a specific record. Record salts can also be combined with the aforementioned attribute salts. Only if the record salt is identical for two records, the same feature (q-gram) will set the same bit positions in the corresponding BFs. However, if the record salts are different, then the probability that the same bit positions are set in the corresponding BFs is very low. Thus, if the attributes (from which the record salts is extracted) contain errors, this will lead to many false-negatives. For this reason only commonly available, stable and small segments of quasi-identifiers, such as year of birth, are suitable as salting key. Consequently, this technique is only an option in PPRL scenarios where the attributes used for salting are guaranteed to be of very high quality which might be rarely the case in practice.

To reduce the aforementioned problem of salting with record-specific keys, we propose to generate the salt by utilizing blocking approaches. Blocking [6] is an essential technique in (privacy-preserving) record linkage to overcome the quadratic complexity of the linkage process since in general each record must be compared to each record of another source. The idea of blocking is to partition records into small blocks and then to compare only records within the same block to reduce the number of record pair comparisons. For this purpose, one or more blocking keys are defined, where each blocking key represents a specific, potentially complex criterion that records must meet to be considered as potential matches. For example, the combination of the first letter of the first and last name and the year of birth might be used as a blocking key. If the attributes used for blocking contain errors, then also the blocking key will be affected leading to many false-negatives, in particular if the blocking key is very restrictive. Hence, often multiple blocking keys are used to increase the probability for records to share at least one blocking key. However, this will lead to duplicate candidates since very similar records will share most blocking keys. The challenge of both, salting and blocking, is to select a key that is as specific as possible (to increase privacy, or to reduce the number of record pair comparisons respectively) and at the same time not prone to errors. For record-dependent salting, only the use of attribute segments was suggested. In contrast, for blocking more sophisticated approaches have been considered, in particular using phonetic codes, e. g., Soundex, or locality-sensitive hashing schemes, e. g., MinHash [4].

3.2.3 Dependency-based Hashing. In traditional record linkage, sophisticated classification models are used to decide whether a record pair represents a match or a non-match. Often these models deploy an attribute-wise or rule-based classification considering the discriminatory power and expected error rate of the attributes [6]. In contrast, PPRL approaches based on record-level BFs only apply classification based on a single similarity threshold since all attributes values are aggregated (encoded) in

a single BF. However, as discussed in Sec. 2.2.1, the record-level BF variant proposed in [9] also considers the weight of attributes by selecting more bits from the field-level BFs of attributes with higher weights. In [32] the authors proposed an extension to the approach of [27] to allow attribute weighting. While also only a single BF is constructed, a different number of hash functions is selected for different attributes according to their weights. Consequently, the higher the weight of an attribute, the more hash functions will be used and thus the more bits the attribute will set in the BF. The idea of varying the number of hash functions can be generalized to dependency-based hashing. For instance, not only the weights of attributes can be considered but instead also the frequency of input features or their position within the attribute value (positional q-grams).

3.3 Bloom Filter Modifications

While the methods described so far modify the way BFs are created, the following approaches are applied directly on the obtained BFs (bit vectors).

3.3.1 Balanced Bloom Filters. Balanced BFs were proposed in [28] for achieving a constant Hamming weight over all BFs. A constant Hamming weight should make the elimination of infrequent patterns more difficult. Balanced BFs are constructed by concatenating a BF with a negative copy of itself and then permuting the underlying bits. For instance, the BF [10011001] will give [10011001] · [01100110] = [1001100101100110] before applying the permutation. Since the size of the BFs is doubled, balanced BFs will increase computing time and required memory for BF comparisons.

3.3.2 XOR-Folding. XOR-folding of bit vectors is a method originating from chemo-informatics to speed up databases queries. In [29] the authors adopted this idea for Bloom-filter-based PPRL for preventing bit pattern attacks. To apply the XOR-folding a BF is split into halves and then the two halves are combined by the XOR-operation. For instance, the BF [11000101] will give [1100] ⊕ [0101] = [1001]. The folding process may be repeated several times. Since the size of the BFs is halved, XOR-folding will decrease computing time and required memory for BF comparisons. The initial evaluation in [29] using unrealistic datasets with full overlap and low error-rates shows that one-time folding does not significantly affect linkage quality. However, n-time folding drastically increases the number of false-positives.

3.3.3 Rule90. In [30] the use of the so-called *Rule90* was suggested to increase the resistance of BFs against bit-pattern-based attacks. The Rule90 is also based on the XOR-operation which is applied on the two neighboring values of each BF bit. Consequently, there are 8 possible combinations (patterns), which are listed in Tab. 2. So each bit b_i ($0 \leq i \leq m-1$) is replaced by the result of XOR-ing the two adjacent bits at positions $(i-1) \bmod m$ and $(i+1) \bmod m$. By using the modulo function the first and the last bit are treated as if they were adjacent. For example, applying Rule90 to the BF [11000101] will lead to the following patterns 1**1**1, 1**1**0, 1**0**0, 0**0**0, 0**0**1, 0**1**0, 1**0**1, 0**1**1 where the middle bit corresponds to the bit at position $i \in \{0, m-1\}$ of the BF. After applying the transformation rules (Tab. 2) we obtain [01101001].

Table 2: Transformation rules for Rule90.

Pattern	111	110	101	100	011	010	001	000
New Bit Value	0	1	0	1	1	0	1	0

3.3.4 Re-hashing. The idea of re-hashing [25] is to use consecutive bits of a BF to generate a new bit vector. Therefore, a window of width w bits is moved over the BF where in each step the window slides forward s positions (step size). At first, a new bit vector v of size m' is allocated. Then, the w bits, which are currently covered by the window, are represented as an integer value. The integer value is then used in combination with a secret key as input for a random number generator (RNG). With that, r new integer values are generated with replacement, each in the range $[0, m'-1]$. Finally, the bits at these r positions are set to one in the bit vector v. For example, given the BF [11000101] and setting $w = 4, s = 2$ will lead to three windows, namely $w_1 = [1100], w_2 = [0001], w_3 = [0101]$. By transforming the bits in each window into an integer value we obtain the seeds 12, 1 and 5. Setting $r = 2$ the RNG might generate the positions $(4, 2), (2, 5), (8, 6)$ for the respective seeds which finally results in the bit vector [001011101]. The evaluation in [28] uses very unrealistic datasets (full overlap, no errors) and shows no clear trend. However, this technique is highly dependent on the choice of the parameters m', w, s and r as well as on the original BFs, in particular the average fill factor (amount of 1-bits).

3.3.5 Random Noise. In order to make the frequency distribution of BFs more uniform, random noise can be added to the BFs [25, 28]. Trivial options are to randomly set bits to one/zero or to flip bits (complement). Additionally, the amount of random noise can depend on the frequency of mapped record features. For instance, for BFs containing frequent q-grams more noise can be added. In [1] a ϵ-differential private BF variant, called BLoom-and-flIP (BLIP), based on permanent randomized response is proposed. Each bit position $b_i, \forall i \in \{0, \ldots, m-1\}$ is assigned a new value b'_i based on the probability f such that

$$b'_i = \begin{cases} 1 & \text{with probability } \frac{1}{2}f \\ 0 & \text{with probability } \frac{1}{2}f \\ b_i & \text{with probability } 1-f. \end{cases}$$

3.3.6 Fake Injections. Another option to modify the frequency distribution of BFs is to add artificial records or attribute values [16]. By inserting random strings containing rarely occurring q-grams the overall frequency distribution will become more uniform making any frequency alignment less accurate. The drawback of fake records is that they produce computational overhead in the matching process. Moreover, it is possible that a fake record will match with another record by chance. Thus, after the linkage, fake records need to be winnowed.

4 BLOOM FILTER PRIVACY MEASURES

Several attacks on BFs have been described in the literature (see Sec. 2.2.2), which show that BFs carry the risk of re-identification of attribute values and even complete records. Currently, the privacy of BF-based encoding schemes is mainly evaluated by simulating attacks and inspecting their results, i. e., the more attribute values and records can be correctly re-identified by an attack, the lower is the assumed degree of privacy of the encoding scheme. However, this way of measuring privacy strongly depends on the used attacks, their assumptions and the used reference dataset. Besides, only a few studies investigated evaluation measures for privacy [33]. These measures are either calculating the probability of suspicion [32] or are based on entropy and information gain between masked and unmasked data [28]. The disadvantage of these measures is that they strongly depend on

the reference data set used. In the following, we therefore propose privacy measures that solely depend on a BF dataset.

To evaluate the disclosure risk of BF-based encoding schemes we propose to analyze the frequency distribution of the BF 1-bits. As described in Sec. 2.2.2, attacks on BFs mostly try to align the frequency of frequent (co-occuring) bit patterns to frequent (co-occuring) record features (q-grams). Thus, the more uniform the frequency distribution of 1-bits is, the less likely an attack will be successful. To measure the uniformity of the bit frequency distribution of a BF dataset $\mathfrak{B}$, we calculate for each BF bit position (column) $0 \leq i < m-1$ the number of 1-bits, given as $\mathbf{c}_i = \sum_{\text{Bf} \in \mathfrak{B}} \text{Bf}(i)$, where $\text{Bf}(i)$ returns the BFs bit value at position i. The total number of 1-bits is then $\mathbf{b} = \sum_{i=0}^{m-1} c_i = \sum_{\text{Bf} \in \mathfrak{B}} |\text{Bf}|$ where $|\text{Bf}|$ denotes the cardinality of a BF (number of 1-bits). We can then calculate for each column its share of the total number of 1-bits, i. e., $\mathbf{p}_i = c_i / b$. Ideally, for a perfect uniform bit distribution, p_i will be close to b/m for all $i \in \{0, m-1\}$.

In mathematics and economics there are several measures that allow to assess the (non-) uniformity of a certain distribution. Consequently, we are adapting the most promising of these measures to our problem. At first, we consider the **Shannon entropy** $\mathcal{H}(\mathfrak{B}) = -\sum_{i=0}^{m-1} p_i \cdot \log_2(p_i)$ since uniform probability will yield maximum entropy. The maximum entropy is given as $\mathcal{H}_{max}(\mathfrak{B}) = \log_2(m)$. We define the normalized Shannon entropy ranging from 0 (high entropy - close to uniform) to 1 (low entropy) as

$$\widetilde{\mathcal{H}}(\mathfrak{B}) = 1 - \frac{H(\mathfrak{B})}{H_{max}(\mathfrak{B})} \tag{1}$$

Next, we consider the **Gini coefficient** [5, 15], which is well-known in economics as a measure of income inequality. The Gini coefficient can range from 0 (perfect equality – all values are the same) to 1 (maximal inequality – one column has all 1-bits and all others have only 0-bits) and is defined as

$$\mathbf{G}(\mathfrak{B}) = \frac{\sum_{i=0}^{m-1} \sum_{j=0}^{m-1} |c_i - c_j|}{2m \cdot b} \tag{2}$$

Moreover, we calculate the **Jensen-Shannon divergence** (JSD) [14] which is a measure of similarity between two probability distributions. The JSD is based on the Kullback-Leibler divergence (KLD) [19], but has better properties for our application: In contrast to the KLD, the JSD is a symmetric measure and the square root of the JSD is a metric known as **Jensen-Shannon distance** (D_{JS}) [10]. For discrete probability distributions P and Q defined on the same probability space, the JSD is defined as

$$\text{JSD}(P \,||\, Q) = \frac{1}{2} \text{KLD}(P \,||\, M) + \frac{1}{2} \text{KLD}(Q \,||\, M) \quad \text{where}$$

$$\text{KLD}(P \,||\, Q) = \sum_{s \in \mathcal{S}} P(s) \cdot \log_2 \left(\frac{P(s)}{Q(s)} \right) \quad \text{and} \quad M = \frac{1}{2}(P + Q).$$

The JSD also provides scores between 0 (identical) to 1 (maximal different). Since we want to measure the uniformity of the bit frequency distribution of a BF dataset $\mathfrak{B}$, we calculate the Jensen-Shannon distance given as

$$\mathbf{D}_{\text{JS}}(\mathfrak{B}) = \sqrt{\text{JSD}(\mathfrak{B})} \tag{3}$$

where

$$\text{JSD}(\mathfrak{B}) = \frac{1}{2} \left(\sum_{i=0}^{m-1} \frac{1}{m} \cdot \log_2 \left(\frac{\frac{1}{m}}{\frac{1}{2} \cdot (p_i + \frac{1}{m})} \right) \right) + \frac{1}{2} \left(\sum_{i=0}^{m-1} p_i \cdot \log_2 \left(\frac{p_i}{\frac{1}{2} \cdot (p_i + \frac{1}{m})} \right) \right)$$

Finally, we measure how many different record features (q-grams) are mapped to each bit position, which we denote as **feature ratio** (fr). The more features are mapped to each position, the harder becomes a one-to-one assignment between bit positions and record features which will limit the accuracy of an attack.

5 EVALUATION SETUP

Before presenting the evaluation results we describe our experimental setup as well as the datasets and metrics we use.

5.1 PPRL Setup

We implement the PPRL process as a three-party protocol assuming a trusted linkage unit [31]. Furthermore, we set the BF length $m = 1024$. To overcome the quadratic complexity of linkage, we use LSH-based blocking based on the Hamming distance [13]. We empirically determined the necessary parameters leading to high efficiency and effectiveness. As a result, we set $\Psi = 16$ (LSH key length) and $\Lambda = 30$ (number of LSH keys) as default. Finally, we calculate the Jaccard coefficient to determine the similarity of candidate record pairs. We classify every record pair with a similarity equal or greater than t as a match. Finally, we apply a one-to-one matching constraint, i. e., a record of one source can match to at maximum one record of another source, utilizing a symmetric best match approach [12].

5.2 Datasets

For evaluation, we use two real datasets that are obtained from the North Carolina voter registration database (NCVR) (https://www.ncsbe.gov/) and the Ohio voter files (OHVF) (https://www.ohiosos.gov/). For both datasets, we select subsets of two snapshots at different points in time. Due to the time difference records contain errors and inconsistencies, e. g., due to marriages/divorces or moves. Please note that we do not insert artificial errors or otherwise modify the records. We only determine how many attributes of a record have changed and use this information to construct subsets with a specific amount of records containing errors. An overview of all relevant dataset characteristics is given in Tab. 3. Each dataset consists of two subsets, S_A and S_B, to be linked with each other. The two subsets are associated with two data owners (or sources) A and B respectively.

Table 3: Dataset characteristics

Characteristic	Dataset	
	N	O
Type	Real (NCVR)	Real (OHVF)
$\lvert S_A \rvert$	50 000	120 000
$\lvert S_B \rvert$	50 000	80 000
$\lvert S_A \cap S_B \rvert$	10 000	40 000
Attributes	{First, middle, last} name, year of birth (YOB), city	{First, middle, last} name, date of birth (BD), city
|Errors|/record	0 (40 %), 1 (30 %), 2 (20 %), 3 (10 %)	0 (37.5 %), 1 (55 %), 2 (6.875 %), 3 (0.625 %)

5.3 Metrics

To assess the linkage quality we determine **recall**, **precision** and **F-measure** (F1-score). Recall measures the proportion of true-matches that have been correctly classified as matches after the linkage process. Precision is defined as the fraction of classified matches that are true-matches. F-measure is the harmonic mean of recall and precision. To assess the privacy (security) of the

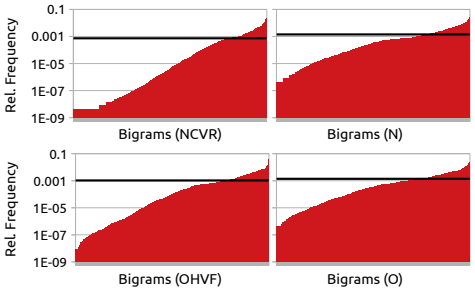
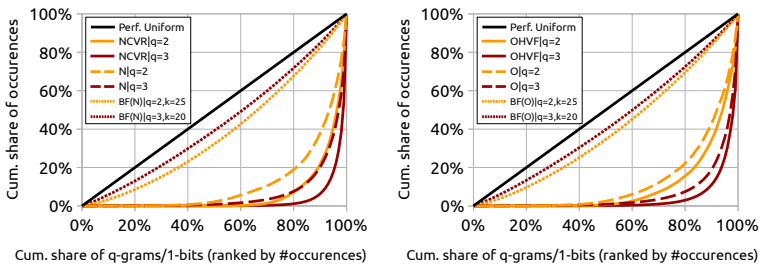


Figure 2: Relative bigram frequencies for used datasets.



(a) NCVR **(b) OHVF**

Figure 3: Comparison of Lorenz curves for plaintext and Bloom filters.

different Bloom-filter-based encoding schemes, we analyze the frequency distribution of the BF's 1-bits in order to determine the **normalized Shannon entropy**, the **Gini coefficient** and the **Jensen-Shannon distance** (see Sec. 4). Furthermore, we calculate the **feature ratio** (fr) that determines how many record features are mapped on average to each bit position.

5.4 Q-Gram Frequencies

Before we begin our evaluation on BFs, we analyze the plaintext frequencies of our datasets N and O as well as the complete NCVR and OHVF datasets. At first, we measure the relative bigram frequencies as shown in Fig. 2. What can be seen in this figure is the high dispersion of bigrams. For the complete NCVR and OHVF the non-uniformity is a bit higher than in our datasets which is mainly due to the larger number of infrequent bigrams. Since our datasets are only subsets from the respective voter registrations (NCVR/OHVF), some of these rare bigrams do simply not occur in our dataset subsets. In Fig. 3 we plot the Lorenz curves [15] for the plaintext datasets as well as BFs (see Sec. 6). These diagrams again illustrate the high dispersion for the plaintext values. Comparing bigrams and trigrams, it can be seen that the non-uniformity for trigrams is even higher than for bigrams. Our observations are confirmed by our uniformity (privacy) measures (see Sec. 4) which we calculate for the datasets as listed in Tab. 4. We use these values as a baseline for the BF privacy analysis. The closer the values for a set of BFs are to these values, the more likely a frequency alignment will be successful. On the other hand, the larger the difference between the values for plaintext and BFs, the better the BFs can hide the plaintext frequencies and thus the less likely a successful frequency alignment becomes. Comparing our three measures, it can be seen that the values for the normalized Shannon entropy ($\widetilde{\mathcal{H}}$) are much lower than the values for the Gini coefficient (G) and the Jensen-Shannon distance (D_{JS}). However, all measures clearly indicate the differences in the frequency distribution of bigrams and trigrams. Comparing both datasets, it can be seen that the non-uniformity of bi- and trigrams is slightly higher for the NCVR than for the OHVF dataset.

Table 4: Analysis of q-gram frequency distribution.

Measure	Datasets							
	Bigrams				Trigrams			
	N	NCVR	O	OHVF	N	NCVR	O	OHVF
$\widetilde{\mathcal{H}}$	0.1848	0.2497	0.1670	0.2027	0.2151	0.2781	0.2142	0.2491
G	0.7709	0.8728	0.7466	0.8047	0.8705	0.9425	0.8729	0.9189
D_{JS}	0.6315	0.7516	0.6107	0.6724	0.7340	0.8392	0.7362	0.8007

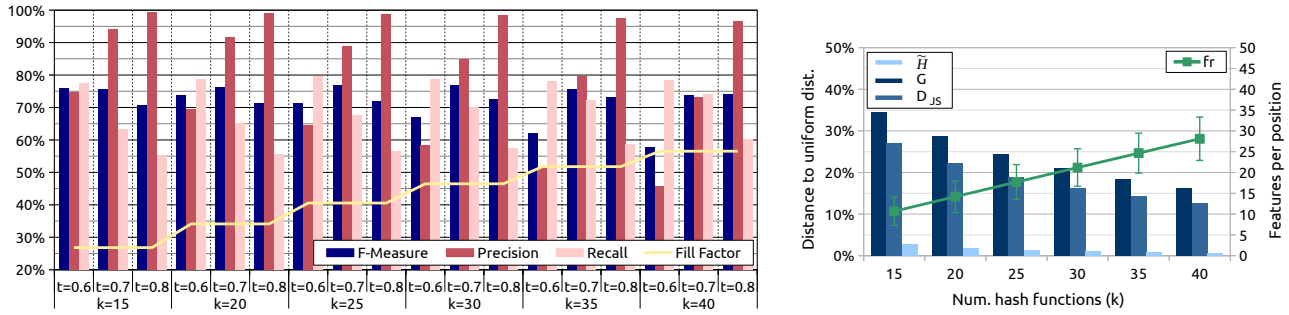
6 RESULTS AND DISCUSSION

In this section, we evaluate various BF variants and hardening techniques in terms of linkage quality and privacy (security).

6.1 Hash Functions and Fill Factor

In the following, we evaluate the linkage quality outcome and the privacy properties of basic BFs by inspecting the frequency distribution of the BF 1-bits compared to the q-gram frequencies. At first, we vary the number of hash functions (k), selecting $k \in \{15, 20, \ldots, 40\}$ and bigrams ($q = 2$), to adjust the fill factor (amount of 1-bits) of the BFs. The results for dataset N are depicted in Fig. 4. The results show, that for the high similarity thresholds of $t = 0.8$, all configurations achieve high precision $\geq 96.58\,\%$, but low recall $\leq 60.19\,\%$, leading to a max. F-measure of 74.16 %. For lower similarity thresholds ($t = \{0.7, 0.6\}$), precision is reduced drastically the more hash functions are used. For instance, setting $t = 0.6$ and $k = 15$, the highest precision of 75.93 % is achieved, while for $k = 40$ the precision is only 45.74 %. In contrast, the higher the number of hash functions, the higher the recall. For instance, setting $t = 0.6$ and $k = 15$, the recall is 73.28 %, while for $k = 40$ it increases to 78.51 %. However, the impact on precision is much higher (difference of around 34 %) than on recall (difference of around 5 %). Overall, the configuration with $t = 0.7$ and $k = 25$ achieves the best F-measure of 76.89 %. However, the other configuration except those with a fill factor over 50 % ($k \in \{35, 40\}$) achieve only slightly less F-measure. When averaging precision and recall for each k over all thresholds, the configurations with $k \leq 25$ achieve a mean F-measure of over 75 %, while for larger k it declines from around 74 % for $k = 30$ to around 71 % for $k = 40$.

Next, we analyze our privacy measures, which are depicted in Fig. 4(b). The figure shows that the more hash functions are used (and thus the higher the fill factor of the BFs) the higher is the avg. number of features that are mapped to each bit position. Even for the lowest number of hash functions, on average around 10 different bigrams are mapped to each individual bit position. Compared to the plaintext frequencies (see Tab. 4), we see that basic BFs have a significantly more uniform frequency distribution than the original plaintext dataset. For instance, using $k = 25$ hash functions, we obtain a Gini coefficient of 0.2443 and a Jensen-Shannon distance of 0.1891 compared to 0.7709 and 0.6315 for the unencoded dataset. Although for the Shannon entropy also a difference is visible, i. e., from 0.1848 for plaintext to 0.0137 for BFs setting $k = 25$, the values are in general much



(a) Quality **(b) Privacy**

Figure 4: Evaluation of standard Bloom filters using bigrams without padding for varying number of hash functions (k) on dataset N.

closer to zero and thus less intuitive to compare. As a consequence, in the following, we will focus on the other two privacy measures. Finally, the privacy measures indicate, that the more hash functions are used, the more closer the 1-bit distribution will get to uniform. However, the effect is not linear, such that the privacy gain is continuously getting lower, in particular for $k \geq 30$.

Table 5: Comparison of Bloom filter encodings using bi- and trigrams with and without padding for dataset N.

q	Pad.	k	t	Recall [%]	Prec. [%]	F-Meas. [%]	Mean Recall [%]	Mean Prec. [%]	Mean F-Meas. [%]
2	No	15	0.6	77.34	74.58	75.93	85.21	67.09	75.07
			0.7	63.27	94.12	75.67			
			0.8	54.96	**99.36**	70.77			
		20	0.6	78.71	69.54	73.84			
			0.7	65.21	91.75	76.24			
			0.8	55.64	99.19	71.29			
		25	0.6	**79.51**	64.58	71.27			
			0.7	67.71	88.95	**76.89**			
			0.8	56.50	98.81	71.89			
		30	0.6	78.68	58.51	67.11			
			0.7	70.14	84.80	76.78			
			0.8	57.43	98.37	72.52			
	Yes	10	0.6	81.48	84.06	**82.75**	90.21	**69.20**	78.32
			0.7	64.56	97.71	77.75			
			0.8	55.08	**99.67**	70.95			
		15	0.6	**83.77**	76.18	79.79			
			0.7	68.46	96.17	79.98			
			0.8	56.02	99.53	71.69			
		20	0.6	83.66	66.15	73.88			
			0.7	72.33	93.07	81.40			
			0.8	57.42	99.37	72.78			
3	No	15	0.6	69.83	86.85	77.42	91.12	63.28	74.69
			0.7	59.49	96.99	73.75			
			0.8	53.71	**99.57**	69.78			
		20	0.6	72.30	83.38	**77.45**			
			0.7	60.71	96.19	74.44			
			0.8	54.30	99.54	70.27			
		25	0.6	73.53	79.83	76.55			
			0.7	62.08	94.93	75.07			
			0.8	54.76	99.41	70.62			
		30	0.6	74.18	75.71	74.94			
			0.7	63.64	93.34	75.68			
			0.8	55.30	99.15	71.00			
		35	0.6	**74.25**	71.66	72.93			
			0.7	65.22	91.43	76.13			
			0.8	55.96	98.86	71.47			
	Yes	10	0.6	79.17	91.99	**85.10**	**93.93**	67.30	**78.42**
			0.7	61.77	98.91	76.05			
			0.8	53.87	**99.70**	69.95			
		15	0.6	**80.87**	86.61	83.64			
			0.7	66.74	98.00	79.40			
			0.8	54.82	99.63	70.72			
		20	0.6	80.31	75.42	77.79			
			0.7	71.96	95.60	82.11			
			0.8	56.20	99.48	71.82			

6.2 Choice of q and the Impact of Padding

In Tab. 5 we compare the linkage quality of BFs using different configurations for $q \in \{2, 3\}$ and padding for dataset N. Without the use of padding the best configuration for bigrams, i. e., $k = 25$ and $t = 0.7$, achieves a slightly less F-measure of 76.89 % than the best configuration for trigrams, i. e., $k = 20$ and $t = 0.6$, of 77.45 %. However, considering the mean over all configurations, using bigrams achieves a slightly higher F-measure of 75.07 % compared to 74.69 % for trigrams. Surprisingly, using trigrams results in an overall higher recall but lower precision if we average the results over all configurations. Moreover, the use of padding leads to a higher linkage quality, i. e., the best configurations for bigrams achieves a F-measure of 82.75 % while for trigrams even 85.10 % is attained. Averaged over all configurations, by using padding recall is increased about 5 % for bigrams and around 2.8 % for trigrams. Interestingly, also precision is increased by around 2.11 % for bigrams and around 4.02 % for trigrams. Thus, for both bigrams and trigrams, the mean F-measure can be increased by padding by more than 3 %. We repeat the experiments on dataset O and report the best configurations in Tab. 6. The results

Table 6: Comparison of Bloom filter encodings using bi- and trigrams with and without padding for dataset O.

q	Padding	k	t	Recall [%]	Precision [%]	F-Meas. [%]
2	No	25	0.7	68.17	88.32	76.95
	Yes	10	0.6	93.99	83.17	88.25
3	No	15	0.6	72.68	82.76	77.39
	Yes	10	0.6	95.49	90.14	92.74

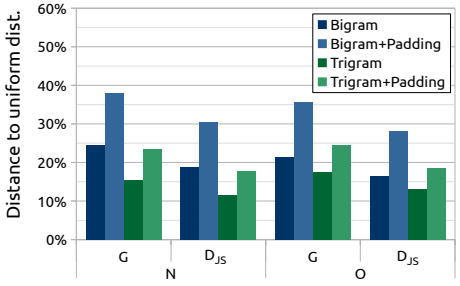
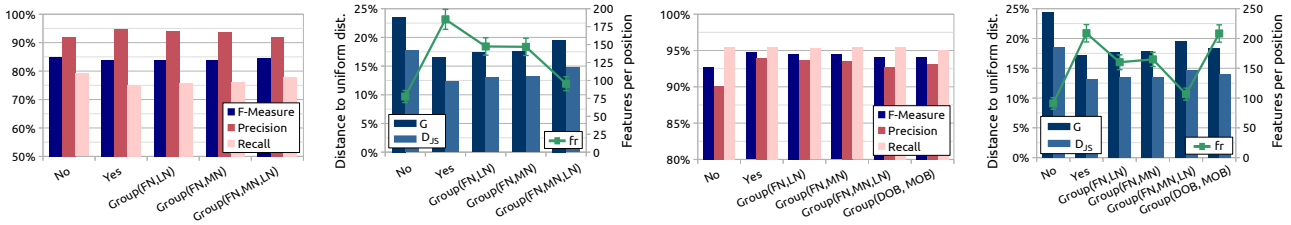


Figure 5: Comparison of Bloom filter privacy for bigrams and trigrams with and without using padding for datasets N and O.



(a) Quality - N dataset **(b) Privacy - N dataset** **(c) Quality - O dataset** **(d) Privacy - O dataset**

Figure 6: Impact of attribute salting.

confirm our previous observation that trigrams with padding lead to the highest linkage quality. Here, the best configuration using trigrams and padding outperforms that with bigrams and padding even slightly more than for dataset N, i. e., F-measure increases 2.35 % for N and 4.49 % for O.

Fig. 5 shows our privacy measures for the best configuration in each group. In general, the use of bigrams leads to a less uniform distribution of 1-bits and thus lower privacy. Also, the use of padding leads to a higher dispersion of the BFs 1-bits. However, even the worst configuration, namely bigrams using padding, leads to a significantly less Gini coefficient as for the plaintext datasets. For N, for instance, the Gini coefficient is reduced from 0.7709 to 0.3801 (see Tab. 4 and Fig. 3). Also the Jensen-Shannon distance reduces drastically, e. g., for dataset N from 0.6315 for the plaintext dataset to 0.3045 for the BF dataset using bigrams with padding. In contrast, the use of trigrams leads to a more even distribution of 1-bits, so that despite using padding, a slightly more uniform frequency distribution is achieved than with bigrams and without using padding.

To summarize, the highest linkage quality is achieved by using padding which indeed leads to less uniform 1-bit distribution making frequency-based cryptanalysis more likely to be successful. However, this can be compensated by using trigrams leading even to a slightly better linkage quality than for bigrams. Consequently, for our following evaluation, we select the best configuration using trigrams and padding with $k = 10$ as a baseline for our experiments.

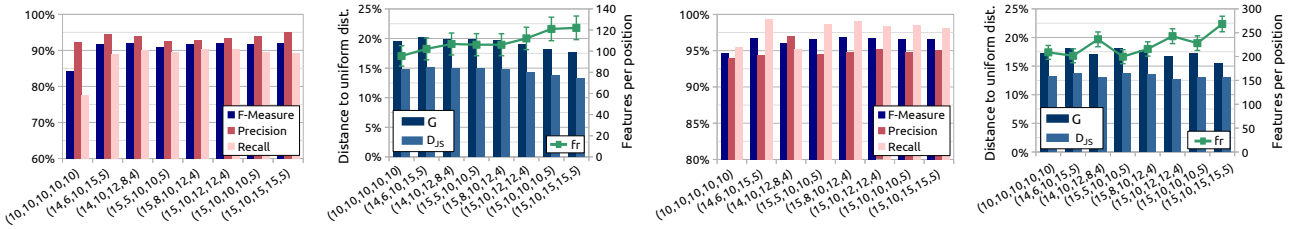
6.3 Salting and Weighting

In this section, we evaluate the impact of methods that alter the BFs hashing process by varying the number of hash functions and using salting keys to modify the hash mapping.

6.3.1 Attribute Salts. Fig. 6 depicts the results for BFs where the used hash functions are keyed (seeded) with a salt depending on the attribute a feature belongs to. For dataset N we observe that using an individual salt for each attribute increases precision from 91.99 % to 94.69 % but also decreases recall from 79.17 % to 75.26 % leading to a F-measure loss of around 1.2 %. Surprisingly, for dataset O precision increases from 90.14 % to 93.93 % while recall remains stable. Simultaneously, the average number of features that are mapped to each bit position increases by more than a factor of two for both datasets (Fig. 6 (b)/(d)). Furthermore, also the Gini coefficient and the Jensen-Shannon distance are significantly decreased and thus indicating an additional smoothing of the 1-bit distribution.

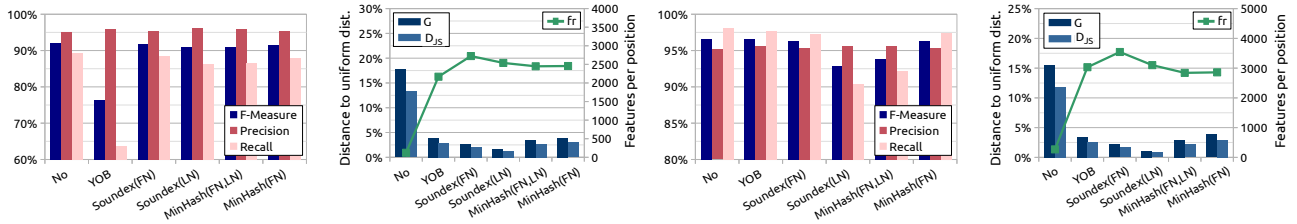
To be tolerant of swapped attributes, we build groups containing name-related attributes, i. e., one group for first name (FN) and last name (LN), one for first name and middle name (MN) and one for all three name components. Additionally, for dataset O, we build a group containing day and month of birth (DOB, MOB). For all attributes within one group, the same attribute salt is used. For dataset N we observe that all groups can slightly increase F-measure, while the group `(FN,MN,LN)` performs best and can increase F-measure to 84.48 %. Compared to the variant without using attribute salts, F-measure is therefore only decreased by 0.6 %. On dataset O, all groups achieve similar results, whereby precision and thus F-measure is always slightly lower than without using groups. Accordingly, swapped attributes seem to occur only rarely in dataset O. Using attribute salt groups also reduce the feature ratio and are also less effective in flattening the 1-bit distribution. Overall, however, the use of attribute salts can significantly reduce the dispersion of 1-bits while maintaining a high linkage quality. Building attribute salt groups can be beneficial for linkage quality, namely for applications where attribute transpositions are likely to occur. In the following, we include attribute salting as a baseline for our experiments, where for dataset N the group `(FN,MN,LN)` is used.

6.3.2 Impact of Attribute Weighting. In the following, we evaluate the impact of attribute weighting. Therefore, the number of hash functions is varied for each attribute depending on attribute weight. We tested several configurations and report the results in Fig. 7. The number of hash functions for each attribute is denoted



(a) Quality - N dataset **(b) Privacy - N dataset** **(c) Quality - O dataset** **(d) Privacy - O dataset**

Figure 7: Evaluation of varying number of hash functions based on attribute weights.



(a) Quality - N dataset **(b) Privacy - N dataset** **(c) Quality - O dataset** **(d) Privacy - O dataset**

Figure 8: Impact of record salting.

in the order (FN,LN,MN,YOB/BD,City). We observe that using attribute weighting strongly affects the linkage quality. All configurations that use a lower number of hash functions to map the attribute *city* can significantly increase both recall and precision. As a consequence, F-measure is improved by more than 6 % to over 91 % for dataset N and by around 2 % to over 96 % for dataset O. Analyzing the privacy results depicted in Fig. 7 (b)/(d), we observe that most weighting configurations can slightly increase the feature ratio and also slightly decrease the non-uniformity of 1-bits. By comparatively analyzing linkage quality and privacy, we select the configuration (15,10,15,15,5) as new baseline since it achieves the highest privacy while F-measure is only minimal less than for (14,10,12,8,4) (dataset N) and (15,10,12,12,4) (dataset O).

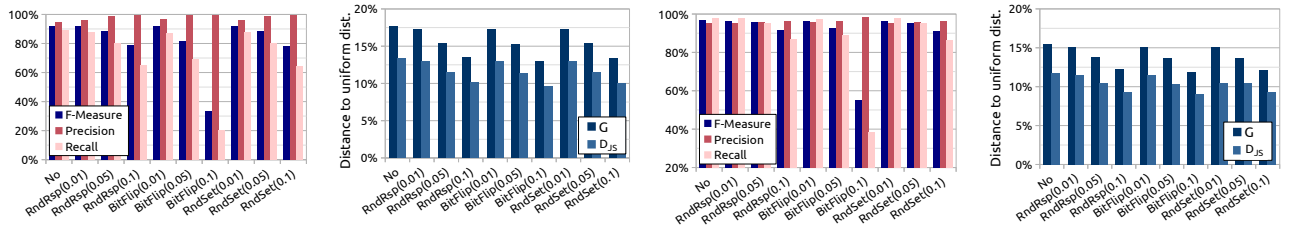
6.3.3 Record Salts. We now evaluate the approach of using a hash function salt that is individually selected for *each record*. The record salt is used in addition to the attribute salt we selected in the previous experiment. We tested several configurations using different attributes (year of birth, first name, last name). As Fig. 8(a)/(c) illustrate, record salts highly affect the linkage quality outcome. If we use the person's year of birth (YOB) as record-specific salt for the BFs hash function, recall drops drastically from 89.20 % (baseline) to only 63.58 % for dataset N. Apparently, in this dataset, this attribute is often erroneous and thus not suitable as record salt. In contrast, applying this configuration on dataset O, recall is only slightly reduced while precision is slightly increased, resulting in nearly the same F-measure. In order to compensate erroneous attributes, we test two techniques that are often utilized as blocking approaches, namely Soundex and MinHashing that we apply on the first and/or last name attribute. All tested approaches can slightly increase precision as they make the hash-mapping of the record features more unique. However, the Soundex and MinHash-based approaches also decrease recall, depending on the attribute(s) used. For instance, using Soundex on last name leads to relatively low recall in both datasets indicating many errors, e. g., due to marriages or divorces. Nevertheless, with the approaches using the first name, a similar high F-measure (loss ≤ 1 %) can be achieved as with the baseline.

Inspecting the privacy results depicted in Fig. 8 (b)/(d), we observe that the number of features that are mapped to each individual bit position is greatly increased by at least a factor of 10. At the same time, using record salts leads to a much more uniform 1-bit distribution. For instance, the Gini coefficient can be reduced from 0.1772 (baseline dataset N) and 0.1549 (baseline dataset O) to less than 0.04 for all tested approaches. The most uniform 1-bit distribution is achieved by using Soundex applied on last name, which leads to a Gini coefficient of less than 0.02. This implies that the 1-bit distribution is almost perfectly uniform which will make any frequency-based attack very unlikely to be successful. By analyzing privacy in relation to quality, we conclude that for both datasets Soundex applied to the first name performs the best and is able to achieve high linkage quality while effectively flattening the 1-bit distribution.

6.4 Modifications

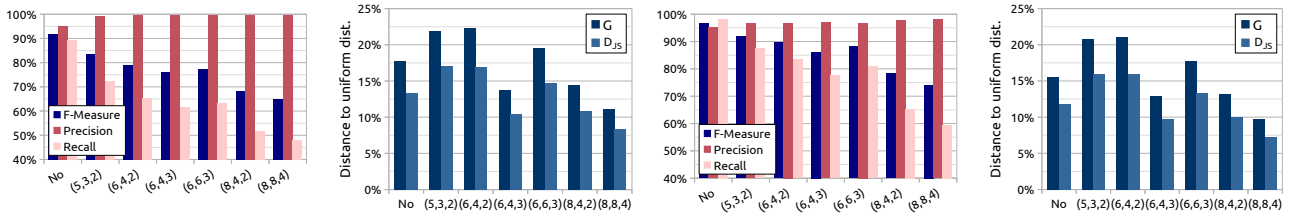
In the following, we evaluate hardening techniques that are applied directly on BFs (bit vectors).

6.4.1 Adding Random Noise. There are several ways of adding random noise to a BF (see Sec. 3.3.5). We compare the randomized response technique (RndRsp), random bit flipping (BitFlip) and randomly setting bits to one (RndSet) with each other. We vary the probability for changing an individual bit by setting $\rho = \{0.01, 0.05, 0.1\}$. The results are depicted in Fig. 9. As expected, recall and F-measure decrease with increasing ρ. While for $\rho = 0.01$ the loss is relatively small, it becomes significantly large for $\rho = 0.1$, in particular for the bit flip approach where recall drastically drops below 20 % for N and below 40 % for O. Interestingly, precision can be raised for all approaches and configurations up to 4.7 % (for $\rho = 0.1$). Overall, the bit flipping approach leads to the highest loss in linkage quality.



(a) Quality - N dataset **(b) Privacy - N dataset** **(c) Quality - O dataset** **(d) Privacy - O dataset**

Figure 9: Evaluation of random noise approaches.



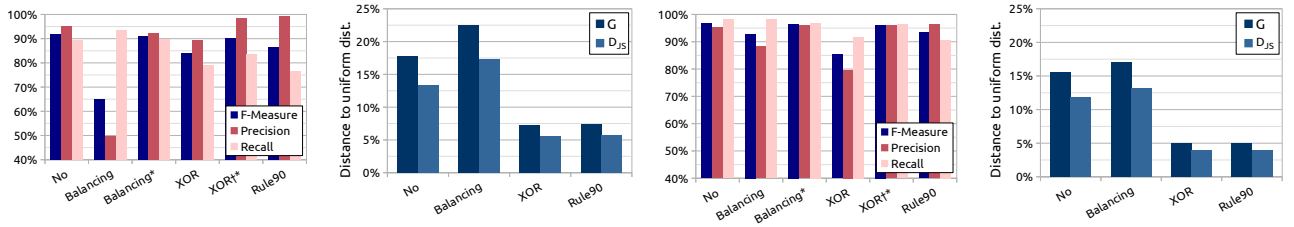
(a) Quality - N dataset **(b) Privacy - N dataset** **(c) Quality - O dataset** **(d) Privacy - O dataset**

Figure 10: Evaluation of re-hashing.

By analyzing the privacy results shown in Fig. 9 (b)/(d), it is evident that all random noise approaches can reduce the frequency information only a little. Only at a high ρ-value of 0.1 the Gini coefficient can be reduced by up to 4.81 % and the Jensen-Shannon distance up to 3.65 %. Analyzing the trade-off between linkage quality and privacy, we observe that a high value for ρ leads to an unacceptable loss in linkage quality. For lower ρ-values both techniques, randomized response and randomly setting bits to one, lead to relatively small losses in linkage quality. However, they are not able to significantly flatten the 1-bit distribution. Nevertheless, hardening techniques based on random noise may impede deterministic attacks by increasing the number of unique bit patterns.

6.4.2 Re-Hashing. To evaluate re-hashing, we tested several configurations regarding window size w, step size s and the number of re-hashed values r. In Fig. 10 we report the best configurations which are denoted in the order `(w,s,r)` setting $m' = m$. The results regarding linkage quality show that re-hashing increases precision but in contrast drastically decreases recall. In general, the larger the window size w the lower the recall that can be achieved. This effect is due to the fact that with larger windows there is a higher probability that a bit in the window is different for two similar BFs. This will result in another integer value (seed) on which the re-hashed 1-bit positions are selected. Even for the configuration with the smallest window size $w = 5$, recall decreases by more than 16 % for both datasets. We could not further decrease the window size, as with $w = 4$ only 16 different bit patterns are possible, so the re-hashed values will be often the same. This observation is confirmed by inspecting the privacy measures illustrated in Fig. 10 (b)/(d). Surprisingly, several configurations, namely `(5,3,2)`, `(6,4,2)` and `(6,6,3)`, will increase the non-uniformity of 1-bits. This is because the re-hashed values will be mapped only to a small range and thus increase the frequencies of these bits. In contrast, the configuration `(6,4,3)` and those with $w = 8$ can flatten the similarity distribution moderately. At the same time, however, these configurations will lead to an unacceptable low recall, e. g., for dataset N to only 61.66 % for `(6,4,3)` or even less then 50 % for `(8,8,4)`. As illustrated by the two configurations `(8,8,4)` and `(8,4,4)`, a reduction of the step size can increase recall since configurations with $s < w$ will lead to overlapping windows and thus a higher chance of finding overlapping bit patterns between two BFs. However, this again increases the unequal distribution of 1-bits. To summarize, we observe that re-hashing will decrease linkage quality while being not effective in increasing the uniformity of 1-bits. Therefore, we can not recommend this method for practical applications.

6.4.3 Balanced Bloom Filter, XOR-folding and Rule90. Finally, we evaluate balanced BFs, XOR-folding and applying Rule90 in terms of linkage quality and privacy. The results are depicted in Fig. 11. The results indicate that balancing reduces precision. While for dataset O precision decreases moderately by 6.93 %, for dataset N it drops drastically by 45.19 %. In contrast, recall remains stable for dataset O whereas it is slightly increased for dataset O. We found that changing our basic similarity threshold from 0.6 to 0.7 can significantly improve linkage quality for balancing. This might be due to the fact that balancing doubles the size of the BFs. Thus, we included the starred version of balancing indicating that a different threshold was used. With this configuration, balancing reduces F-measure only slightly for both datasets. For dataset N this is due to a little less precision and a little higher recall than for the baseline. For dataset O, however, it is the other way around, i. e., less recall and higher precision. XOR-folding also causes a reduction in linkage quality for both datasets. Since XOR-folding halves the size of the BFs, LSH-based blocking is affected in such a way that the amount of bits selected for the LSH keys is comparatively large. We therefore reduced the LSH key length from $\Psi = 16$ to $\Psi = 10$ and indicate this configuration with a dagger (†). By using this configuration and setting $t = 0.7$, for both datasets XOR-folding results in a minor loss of F-measure of less than 1 % compared to the baseline. Primarily accountable for the high F-measure is the high precision, which is slightly increased. Furthermore, we observe that



(a) Quality - N dataset **(b) Privacy - N dataset** **(c) Quality - O dataset** **(d) Privacy - O dataset**

Figure 11: Evaluation of balanced Bloom filters, XOR-folding and Rule90.

applying Rule90 also leads to a relatively high loss of recall of around 12.5 % for datast N and 7.6 % for dataset O. Again, Rule90 increases precision slightly thus leading to a moderate loss of F-measure of around 5.5 % for dataset N and 3.2 % for dataset O.

Examining Fig. 11 (b)/(d), we can see that balancing interestingly increases the dispersion of 1-bits. For dataset N, for instance, the Gini coefficient is increased by around 4.8 % and the Jensen-Shannon distance by around 4 %. In contrast, XOR-folding and Rule90 lead to a more uniform distribution of 1-bits. Both approaches reduce the Gini coefficient by about 10 % and the Jensen-Shannon distance by more than 7.5 %. Considering both, linkage quality and privacy, we conclude that XOR-folding performs the best by maintaining high linkage quality while effectively flattening the 1-bit distribution.

7 CONCLUSION

Bloom filters are frequently used in both research and practice for PPRL applications. In this paper, we reviewed and classified various BF variants and hardening techniques that aim at making Bloom filters more robust against cryptanalysis. Currently, no privacy measure exists that allows comparison of different encoding schemes in terms of privacy (security) and is independent of any reference dataset. We therefore proposed three privacy measures that allow assessing the privacy properties of Bloom filter encodings. These measures are based solely on a set of Bloom filters and do not need any reference dataset or other information. Moreover, we comprehensively evaluated the Bloom filter variants and hardening techniques in terms of both linkage quality and privacy. The evaluation showed that multiple hardening techniques drastically reduce linkage quality and are thus not applicable in real-world scenarios. However, in particular two techniques, namely salting and XOR-folding, drastically reduce any frequency information while maintaining high linkage quality. Carefully selected Bloom filter parameters in combination with these techniques will make any frequency-based cryptanalysis very unlikely to be successful.

For future work, we aim to evaluate these approaches against modern Bloom filter attacks described in the literature to further verify our findings.

REFERENCES

[1] Mohammad Alaggan, Sébastien Gambs, and Anne-Marie Kermarrec. 2012. BLIP: Non-interactive Differentially-Private Similarity Computation on Bloom filters. In *Stabilization, Safety and Security of Distributed Systems*. 202–216. https://doi.org/10.1007/978-3-642-33536-5_20

[2] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *CACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[3] Andrei Broder and Michael Mitzenmacher. 2004. Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1, 4 (2004), 485–509. https://doi.org/10.1080/15427951.2004.10129096

[4] A. Z. Broder. 1998. On the Resemblance and Containment of Documents. In *Compression and Complexity of Sequences*. 21–29. https://doi.org/10.1109/SEQUEN.1997.666900

[5] Lidia Ceriani and Paolo Verme. 2012. The Origins of the Gini Index: Extracts from Variabilità e Mutabilità (1912) by Corrado Gini. *The Journal of Economic Inequality* 10, 3 (2012), 421–443. https://doi.org/10.1007/s10888-011-9188-x

[6] Peter Christen. 2012. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. https://doi.org/10.1007/978-3-642-31164-2

[7] Peter Christen, Thilina Ranbaduge, Dinusha Vatsalan, and Rainer Schnell. 2018. Precise and Fast Cryptanalysis for Bloom Filter Based Privacy-Preserving Record Linkage. *IEEE TKDE* (2018). https://doi.org/10.1109/TKDE.2018.2874004

[8] Peter Christen, Anushka Vidanage, Thilina Ranbaduge, and Rainer Schnell. 2018. Pattern-Mining Based Cryptanalysis of Bloom Filters for Privacy-Preserving Record Linkage. In *PAKDD*. Vol. 10939. Springer, 530–542. https://doi.org/10.1007/978-3-319-93040-4_42

[9] Elizabeth A. Durham, Murat Kantarcioglu, Yuan Xue, Csaba Toth, Mehmet Kuzu, and Bradley Malin. 2014. Composite Bloom Filters for Secure Record Linkage. *IEEE TKDE* 26, 12 (2014), 2956–2968. https://doi.org/10.1109/TKDE.2013.91

[10] D.M. Endres and J.E. Schindelin. 2003. A New Metric for Probability Distributions. *IEEE Transactions on Information Theory* 49, 7 (2003), 1858–1860. https://doi.org/10.1109/TIT.2003.813506

[11] Wendy R. Fox and Gabriel W. Lasker. 1983. The Distribution of Surname Frequencies. *Int. Statistical Review* 51, 1 (1983), 81–87. https://doi.org/10.2307/1402733

[12] Martin Franke, Ziad Sehili, Marcel Gladbach, and Erhard Rahm. 2018. Post-Processing Methods for High Quality Privacy-Preserving Record Linkage. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 263–278. https://doi.org/10.1007/978-3-030-00305-0_19

[13] Martin Franke, Ziad Sehili, and Erhard Rahm. 2018. Parallel Privacy-Preserving Record Linkage Using LSH-Based Blocking. In *IoTBDS*. 195–203. https://doi.org/10.5220/0006682701950203

[14] B. Fuglede and F. Topsoe. 2004. Jensen-Shannon Divergence and Hilbert Space Embedding. In *Int. Symposium on Information Theory (ISIT)*. 31–. https://doi.org/10.1109/ISIT.2004.1365067

[15] Joseph L. Gastwirth. 1972. The Estimation of the Lorenz Curve and Gini Index. *The Review of Economics and Statistics* 54, 3 (1972), 306. https://doi.org/10.2307/1937992

[16] Alexandros Karakasidis, Vassilios S. Verykios, and Peter Christen. 2012. Fake Injection Strategies for Private Phonetic Matching. In *Data Privacy Management and Autonomous Spontaneus Security*. 9–24. https://doi.org/10.1007/978-3-642-28879-1_2

[17] Dimitrios Karapiperis, Aris Gkoulalas-Divanis, and Vassilios S. Verykios. 2018. FEDERAL: A Framework for Distance-Aware Privacy-Preserving Record Linkage. *IEEE TKDE* 30, 2 (2018), 292–304. https://doi.org/10.1109/TKDE.2017.2761759

[18] Martin Kroll and Simone Steinmetzer. 2014. Automated Cryptanalysis of Bloom Filter Encryptions of Health Records. *German Record Linkage Center, Working Paper Series* (2014). https://doi.org/10.2139/ssrn.3530864

[19] S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79–86. https://doi.org/10.1214/aoms/1177729694

[20] Mehmet Kuzu, Murat Kantarcioglu, Elizabeth Durham, and Bradley Malin. 2011. A Constraint Satisfaction Cryptanalysis of Bloom Filters in Private Record Linkage. In *Privacy Enhancing Technologies*. 226–245. https://doi.org/10.1007/978-3-642-22263-4_13

[21] William Mitchell, Rinku Dewri, Ramakrishna Thurimella, and Max Roschke. 2016. A Graph Traversal Attack on Bloom Filter Based Medical Data Aggregation. *Int. Journal of Big Data Intelligence* (2016). https://doi.org/10.1504/IJBDI.2017.086956

[22] Robert Morris and Ken Thompson. 1979. Password Security: A Case History. *Commun. ACM* 22, 11 (1979), 594–597. https://doi.org/10.1145/359168.359172

[23] National Institute of Standards and Technology. 2008. *The Keyed-Hash Message Authentication Code (HMAC)*. Technical Report NIST FIPS 198-1. https://doi.org/10.6028/NIST.FIPS.198-1

[24] Frank Niedermeyer, Simone Steinmetzer, Martin Kroll, and Rainer Schnell. 2014. Cryptanalysis of Basic Bloom Filters Used for Privacy Preserving Record Linkage. *Journal of Privacy and Confidentiality* 6, 2 (2014). https://doi.org/10.29012/jpc.v6i2.640

[25] Rainer Schnell. 2014. Privacy Preserving Record Linkage. In *Methodological Developments in Data Linkage*. 201–225.

[26] Rainer Schnell, Tobias Bachteler, and Jörg Reiher. 2009. Privacy-Preserving Record Linkage Using Bloom Filters. *BMC Medical Informatics and Decision Making* 9, 1 (2009). https://doi.org/10.1186/1472-6947-9-41

[27] Rainer Schnell, Tobias Bachteler, and Jörg Reiher. 2011. A Novel Error-Tolerant Anonymous Linking Code. (2011). https://doi.org/10.2139/ssrn.3549247

[28] Rainer Schnell and Christian Borgs. 2016. Randomized Response and Balanced Bloom Filters for Privacy Preserving Record Linkage. In *IEEE ICDMW*. 218–224. https://doi.org/10.1109/ICDMW.2016.0038

[29] Rainer Schnell and Christian Borgs. 2016. XOR-Folding for Hardening Bloom Filter-Based Encryptions for Privacy-Preserving Record Linkage. *German Record Linkage Center, Working Paper Series* (2016). https://doi.org/10.2139/ssrn.3527984

[30] Rainer Schnell and Christian Borgs. 2018. Hardening Encrypted Patient Names Against Cryptographic Attacks Using Cellular Automata. In *IEEE ICDMW*. 518–522. https://doi.org/10.1109/ICDMW.2018.00082

[31] Dinusha Vatsalan and Peter Christen. 2016. Privacy-Preserving Matching of Similar Patients. *Journal of Biomedical Informatics* 59 (2016), 285–298. https://doi.org/10.1016/j.jbi.2015.12.004

[32] Dinusha Vatsalan, Peter Christen, Christine M. O'Keefe, and Vassilios S. Verykios. 2014. An Evaluation Framework for Privacy-Preserving Record Linkage. *Journal of Privacy and Confidentiality* 6, 1 (2014).

[33] Dinusha Vatsalan, Ziad Sehili, Peter Christen, and Erhard Rahm. 2017. Privacy-Preserving Record Linkage for Big Data: Current Approaches and Research Challenges. In *Handbook of Big Data Technologies*. Springer, 851–895. https://doi.org/10.1007/978-3-319-49340-4_25

Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation

Suyash Gupta Jelle Hellings Sajjad Rahnama Mohammad Sadoghi
Exploratory Systems Lab
Department of Computer Science
University of California, Davis

ABSTRACT

Multi-party data management and blockchain systems require data sharing among participants. To provide resilient and consistent data sharing, transactions engines rely on Byzantine Fault-Tolerant consensus (BFT), which enables operations during failures and malicious behavior. Unfortunately, existing BFT protocols are unsuitable for high-throughput applications due to their high computational costs, high communication costs, high client latencies, and/or reliance on twin-paths and non-faulty clients.

In this paper, we present the *Proof-of-Execution consensus protocol* (PoE) that alleviates these challenges. At the core of PoE are *out-of-order* processing and *speculative execution*, which allow PoE to execute transactions before consensus is reached among the replicas. With these techniques, PoE manages to reduce the costs of BFT in normal cases, while guaranteeing reliable consensus for clients in all cases. We envision the use of PoE in high-throughput multi-party data-management and blockchain systems. To validate this vision, we implement PoE in our efficient ResilientDB fabric and extensively evaluate PoE against several state-of-the-art BFT protocols. Our evaluation showcases that PoE achieves up-to-80% higher throughputs than existing BFT protocols in the presence of failures.

1 INTRODUCTION

In *federate data management* a single common database is managed by many independent stakeholders (e.g., an industry consortium). In doing so, federated data management can ease data sharing and improve data quality [17, 32, 48]. At the core of federated data management is *reaching agreement* on any updates on the common database in an efficient manner, this to enable fast query processing, data retrieval, and data modifications.

One can achieve federated data management by *replicating* the common database among all participant, this by replicating the sequence of transactions that affect the database to all stakeholders. One can do so using commit protocols designed for distributed databases such as two-phase [22] and three-phase commit [49], or by using crash-resilient replication protocols such as Paxos [39] and Raft [45].

These solutions are error-prone in a federated *decentralized* environment in which each stakeholder manages its own replicas and replicas of each stakeholder can fail (e.g., due to software, hardware, or network failure) or act malicious: commit protocols and replication protocols can only deal with crashes. Consequently, recent federated designs propose the usage of Byzantine Fault-Tolerant (BFT) consensus protocols. BFT consensus aims at *ordering client requests among a set of replicas, some of which could be Byzantine, such that all non-faulty replicas reach agreement on a common order for these requests* [9, 21, 29, 38, 51]. Furthermore, BFT consensus comes with the added benefit of *democracy*, as BFT consensus gives all replicas an equal vote in all agreement decisions, while the resilience of BFT can aid in dealing with the billions of dollars losses associated with prevalent attacks on data management systems [44].

Akin to commit protocols, the majority of BFT consensus protocols use a *primary-backup model* in which one replica is designated *the primary* that coordinates agreement, while the remaining replicas act as backups and follow the protocol [46]. This primary-backup BFT consensus was first popularized by the influential PBFT consensus protocol of Castro and Liskov [9]. The design of PBFT requires at least $3\mathbf{f}+1$ replicas to deal with up-to-$\mathbf{f}$ malicious replicas and operates in *three communication phases*, two of which necessitate quadratic communication complexity. As such, PBFT is considered costly when compared to commit or replication protocols, which has negatively impacted the usage of BFT consensus in large-scale data management systems [8].

The recent interest in blockchain technology has revived interest in BFT consensus, has led to several new resilient data management systems (e.g., [3, 18, 29, 43]), and has led to the development of new BFT consensus protocols that promise efficiency at the cost of flexibility (e.g., [21, 28, 38, 51]). Despite the existence of these modern BFT consensus protocols, the majority of BFT-fueled systems [3, 18, 29, 43] still employ the classical time-tested, flexible, and safe design of PBFT, however.

In this paper, we explore different design principles that can enable implementing a scalable and reliable agreement protocol that shields against malicious attacks. We use these design principles to introduce Proof-of-Execution (PoE), a novel BFT protocol that achieves resilient agreement in just three linear phases. To concoct PoE's scalable and resilient design, we start with PBFT and successively add *four* design elements:

(I1) **Non-Divergent Speculative Execution.** In PBFT, when the primary replica receives a client request, it forwards that request to the backups. Each backup on receiving a request from the primary agrees to support by broadcasting a PREPARE message. When a replica receives PREPARE message from the majority of other replicas, it marks itself as *prepared* and broadcasts a COMMIT message. Each replica that has prepared, and receives COMMIT messages from a majority of other replicas, executes the request.

Evidently, PBFT requires two phases of *all-to-all* communication. Our first ingredient towards faster consensus is speculative execution. In PBFT terminology, PoE replicas execute requests after they get *prepared*, that is, they do not broadcast COMMIT messages. This speculative execution is non-divergent as each replica has a partial guarantee–it has prepared–prior to execution.

(I2) **Safe Rollbacks and Robustness under Failures.** Due to speculative execution, a malicious primary in PoE can ensure that only a subset of replicas prepare and execute a request. Hence, a client may or may not receive a sufficient number of matching responses. PoE ensures that if a client receives a *full proof-of-execution*, consisting of responses from a majority of the non-faulty replicas, then such a request persists in time. Otherwise, PoE permits replicas to *rollback* their state if necessary. This proof-of-execution is the cornerstone of the correctness of PoE.

(I3) **Agnostic Signatures and Linear Communication.** BFT protocols are run among distrusting parties. To provide security, these protocols employ cryptographic primitives for signing the messages and generating message digests. Prior works have shown that the choice of cryptographic signature scheme can impact the performance of the underlying system [9, 30]. Hence, we allow replicas to either employ *message authentication codes* (MACs) or *threshold signatures* (TSs) for signing [36]. When few replicas are participating in consensus (up to 16), then a single phase of all-to-all communication is inexpensive and using MACs for such setups can make computations cheap. For larger setups, we employ TSs to achieve linear communication complexity. TSs permit us to split a phase of all-to-all communication into *two linear phases* [21, 51].

(I4) **Avoid Response Aggregation.** SBFT [21], a recently-proposed BFT protocol, suggests the use of a single replica (designated as the *executor*) to act as a response aggregator. In specific, all replicas execute each client request and send their response to the executor. It is the duty of the executor to reply to the client and *send a proof* that a majority of the replicas not only executed this request, but also outputted the same result. In PoE, we avoid this additional communication between the replicas by allowing each replica to respond directly to the client.

In specific, we make the following contributions:

(1) We introduce PoE, a novel Byzantine fault-tolerant consensus protocol that uses *speculative execution* to reach agreement among replicas.
(2) To guarantee failure recovery in the presence of speculative execution and Byzantine behavior, we introduce a novel view-change protocol that can rollback requests.
(3) PoE supports batching, out-of-order processing, and is signature-scheme agnostic and can be made to employ either MACs or threshold signatures.
(4) PoE does not rely on non-faulty replicas, clients, or trusted hardware to achieve safe and efficient consensus.
(5) To validate our vision of using PoE in resilient federated data management systems, we implement PoE and four other BFT protocols (Zyzzyva, Pbft, SBFT, and HotStuff) in our efficient ResilientDB[1] fabric [23–25, 27, 29, 30, 47].
(6) We extensively evaluate PoE against these protocols on a Google Cloud deployment consisting of 91 replicas and 320 k clients under (i) no failure, (ii) backup failure, (iii) primary failure, (iv) batching of requests, (v) zero payload, and (vi) scaling the number of replicas. Further, to prove the correctness of our results, we also stress test PoE and other protocols in a simulated environment. Our results show that PoE can achieve up to 80% more throughput than existing BFT protocols in the presence of failures.

To the best of our knowledge, PoE is the first protocol that achieves consensus in *only two phases* while being able to deal with Byzantine failures and without relying on trusted clients (e.g., Zyzzyva [38]) or on trusted hardware (e.g., MinBFT [50]). Hence, PoE can serve as a drop-in replacement of Pbft to improve scalability and performance in permissioned blockchain fabrics such as our ResilientDB fabric [27–31], MultiChain [20], and Hyperledger Fabric [4]; in multi-primary meta-protocols such as RCC [26, 28]; and in sharding protocols such as AHL [15].

2 ANALYSIS OF DESIGN PRINCIPLES

To arrive at an optimal design for PoE, we studied practices followed by state-of-the-art distributed data management systems and applied their principles to the design of PoE where possible. In Figure 1, we present a comparison of PoE against *four* well-known resilient consensus protocols.

Protocol	Phases	Messages	Resilience	Requirements
Zyzzyva	1	$O(\mathbf{n})$	0	Reliable clients and unsafe
PoE (our paper)	3	$O(3\mathbf{n})$	$\mathbf{f}$	Sign. agnostic
Pbft	3	$O(\mathbf{n} + 2\mathbf{n}^2)$	$\mathbf{f}$	
HotStuff	8	$O(8\mathbf{n})$	$\mathbf{f}$	Sequential Consensus
SBFT	5	$O(5\mathbf{n})$	0	Optimistic path

Figure 1: Comparison of BFT consensus protocols in a system with n replicas of which f are faulty. The costs given are for the normal-case behavior.

To illustrate the merits of PoE's design, we first briefly look at Pbft. The last phase of Pbft ensures that non-faulty replicas only execute requests and inform clients when there is a guarantee that such a transaction will be recovered after any failures. Hence, clients need to wait for only $\mathbf{f} + 1$ identical responses, of which at-least one is from a non-faulty replica, to ensure *guaranteed execution*. By eliminating this last phase, replicas speculatively execute requests before obtaining recovery guarantees. This impacts Pbft-style consensus in two ways:

(1) First, clients need a way to determine *proof-of-execution* after which they have a guarantee that their requests are executed and maintained by the system. We shall show that such a proof-of-execution can be obtained using $\mathbf{nf} \geq 2\mathbf{f} + 1$ identical responses (instead of $\mathbf{f} + 1$ responses).
(2) Second, as requests are executed before they are guaranteed, replicas need to be able to rollback requests that are dropped during periods of recovery.

PoE's speculative execution guarantees that requests with a proof-of-execution will never rollback and that only a single request can obtain a proof-of-execution per round. Hence, speculative execution provides the same strong consistency (safety) of Pbft in all cases, this at much lower cost under normal operations. Furthermore, we show that speculative execution is fully compatible with other scalable design principles applied to Pbft, e.g., batching and out-of-order processing to maximize throughput, even with high message delays.

Out-of-order execution. Typical BFT systems follow the *order-execute* model: first replicas agree on a unique order of the client request, and only then they execute the requests in order [9, 21, 29, 38, 51]. Unfortunately, this prevents these systems from providing any support for concurrent execution. A few BFT systems suggest executing prior to ordering, but even such systems need to re-verify their results prior to committing changes [4, 35]. Our PoE protocol lies between these two extremes: the replicas speculatively execute using only partial ordering guarantees. By doing so, PoE can eliminate communication costs and minimize latencies of typical BFT systems, this without needing to re-verify results in the normal case.

Out-of-order processing. Although BFT consensus typically executes requests in-order, this does not imply they need to process proposals to order requests sequentially. To maximize throughput, Pbft and other primary-backup protocols support *out-of-order processing* in which all available bandwidth of the primary is used to continuously propose requests (even when previous proposals are still being processed by the system). By doing so, out-of-order processing can eliminate the impact of high message delays. To provide out-of-order processing, all replicas will process any request proposed as the k-th request whenever k is within some *active window* bounded by a *low-watermark* and *high-watermark* [9]. These watermarks are increased as the

[1] ResilientDB is open-sourced at https://github.com/resilientdb.

system progresses. The size of this active window is—in practice—only limited by the memory resources available to replicas. As out-of-order processing is an essential technique to deliver high throughputs in environments with high message delays, we have included out-of-order processing in the design of PoE.

Twin-path consensus. Speculative execution employed by PoE is different that the *twin-path model* utilized by Zyzzyva [38] and SBFT [21]. These twin-path protocols have an optimistic *fast* path that works only if none of the replicas are *faulty* and require aid to determine whether these optimistic condition hold.

In the fast path of Zyzzyva, primaries propose requests, and backups directly execute such proposals and inform the client (without further coordination). The client waits for responses from all $\mathbf{n}$ replicas before marking the request executed. When the client does not receive $\mathbf{n}$ responses, it *timeouts* and sends a message to all replicas, after which the replicas perform an expensive client-dependent *slow-path* recovery process (which is prone to errors when communication is unreliable [2]).

The fast path of SBFT can deal with up to $\mathbf{c}$ crash-failures using $3\mathbf{f} + 2\mathbf{c} + 1$ replicas and uses threshold signatures to make communication linear. The fast path of SBFT requires a reliable collector and executor to aggregate messages and to send only *a single* (instead of at-least-$\mathbf{f} + 1$) response to the client. Due to aggregating execution, the fast path of SBFT still performs four rounds of communication before the client gets a response, whereas PoE only uses two rounds of communication (or three when PoE uses threshold signatures). If the fast path *timeouts* (e.g., the collector or executor fails), then SBFT falls back to a threshold-version of Pbft that takes an additional round before the client gets a response. Twin-path consensus is in sharp contrast with the design of PoE, which does not need outside aid (reliable clients, collectors, or executors), and can operate optimally even while dealing with replica failures.

Primary rotation. To minimize the influence of any single replica on bft consensus, HotStuff opts to replace the primary after every consensus decision. To efficiently do so, HotStuff uses an extra communication phase (as compared to Pbft), which minimizes the cost of primary replacement. Furthermore, HotStuff uses threshold signatures to make its communication linear (resulting in eight communication phases before a client gets responses). The event-based version of HotStuff can overlap phases of consecutive rounds, thereby assuring that consensus of a client request starts in every one-to-all-to-one communication phase. Unfortunately, the primary replacements require that all consensus rounds are performed in a strictly *sequential* manner, eliminating any possibility of *out-of-order processing*.

3 PROOF-OF-EXECUTION

In our *Proof-of-Execution consensus protocol* (PoE), the primary replica is responsible for proposing transactions requested by clients to all backup replicas. Each backup replica *speculatively* executes these transactions with the belief that the primary is behaving correctly. Speculative execution expedites processing of transactions in all cases. Finally, when malicious behavior is detected, replicas can recover by *rolling back transactions*, which ensures correctness without depending on any twin-path model.

3.1 System model and notations

Before providing a full description of our PoE protocol, we present the system model we use and the relevant notations.

A system is a set $\mathfrak{R}$ of *replicas* that process client requests. We assign each replica $\mathrm{R} \in \mathfrak{R}$ a unique identifier $\mathrm{id}(\mathrm{R})$ with $0 \leq \mathrm{id}(\mathrm{R}) < |\mathfrak{R}|$. We write $\mathcal{F} \subseteq \mathfrak{R}$ to denote the set of *Byzantine replicas* that can behave in arbitrary, possibly coordinated and malicious, manners. We assume that non-faulty replicas (those in $\mathfrak{R} \setminus \mathcal{F}$) behave in accordance to the protocol and are deterministic: on identical inputs, all non-faulty replicas must produce identical outputs. We do not make any assumptions on clients: all client can be malicious without affecting PoE. We write $\mathbf{n} = |\mathfrak{R}|$, $\mathbf{f} = |\mathcal{F}|$, and $\mathbf{nf} = |\mathfrak{R} \setminus \mathcal{F}|$ to denote the number of replicas, faulty replicas, and non-faulty replicas, respectively. We assume that $\mathbf{n} > 3\mathbf{f}$ ($\mathbf{nf} > 2\mathbf{f}$).

We assume *authenticated communication*: Byzantine replicas are able to impersonate each other, but replicas cannot impersonate non-faulty replicas. Authenticated communication is a minimal requirement to deal with Byzantine behavior. Depending on the type of message, we use message authentication codes (MACs) or threshold signatures (TSs) to achieve authenticated communication [36]. MACs are based on symmetric cryptography in which every pair of communicating nodes has a *secret key*. We expect non-faulty replicas to keep their *secret keys* hidden. TSs are based on asymmetric cryptography. In specific, each replica holds a distinct *private key*, which it can use to create a signature share. Next, one can produce a valid threshold signature given at least $\mathbf{nf}$ such signature shares (from distinct replicas). We write $s\langle v \rangle_i$ to denote the signature share of the i-th replica for signing value v. Anyone that receives a set $T = \{s\langle v \rangle_j \mid j \in T'\}$ of signature shares for v from $|T'| = \mathbf{nf}$ distinct replicas, can aggregate T into a single signature $\langle v \rangle$. This digital signature can then be verified using a public key.

We also employ a *collision-resistant cryptographic hash function* $\mathrm{D}(\cdot)$ that can map an arbitrary value v to a constant-sized digest $\mathrm{D}(v)$ [36]. We assume that it is practically impossible to find another value v', $v \neq v'$, such that $\mathrm{D}(v) = \mathrm{D}(v')$. We use notation $v||w$ to denotes the *concatenation* of two values v and w.

Next, we define the consensus provided by PoE.

Definition 3.1. A single run of any *consensus protocol* should satisfy the following requirements:

Termination. Each non-faulty replica executes a transaction.

Non-divergence. All non-faulty replicas execute the same transaction.

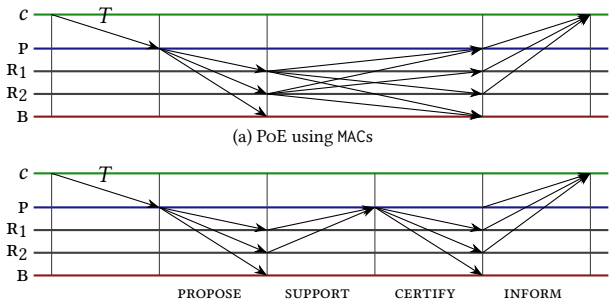
Termination is typically referred to as *liveness*, whereas non-divergence is typically referred to as *safety*. In PoE, execution is speculative: replicas can execute and rollback transactions. To provide safety, PoE provides speculative non-divergence instead of non-divergence:

Speculative non-divergence. If $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas accept and execute the same transaction T, then all non-faulty replicas will eventually accept and execute T (after rolling back any other executed transactions).

To provide *safety*, we do not need any other assumptions on communication or on clients. Due to well-known impossibility results for asynchronous consensus [19], we can only provide *liveness* in periods of *reliable bounded-delay communication* during which all messages sent by non-faulty replicas will arrive at their destination within some maximum delay.

3.2 The Normal-Case Algorithm of PoE

PoE operates in *views* $v = 0, 1, \ldots$. In view v, replica R with $\mathrm{id}(\mathrm{R}) = v \bmod \mathbf{n}$ is elected as the primary. The design of PoE relies on authenticated communication, which can be provided using MACs or TSs. In Figure 2, we sketch the normal-case working of PoE for both cases. For the sake of brevity, we will describe PoE built on top of TSs, which results in a protocol with low—*linear*—message complexity in the normal case. The full pseudo-code for



(a) PoE using MACs

(b) PoE using TSs.

Figure 2: Normal-case algorithm of PoE: Client c sends its request containing transaction T to the primary P, which proposes this request to all replicas. Although replica B is Byzantine, it fails to affect PoE.

this algorithm can be found in Figure 3. In Section 3.6, we detail the minimal changes to PoE necessary when switching to MACs.

Consider a view v with primary P. To request execution of transaction T, a client c signs transaction T and sends the signed transaction $\langle T \rangle_c$ to P. The usage of signatures assures that malicious primaries cannot forge transactions. To initiate replication and execution of T as the k-th transaction, the primary proposes T to all replicas via a PROPOSE message.

After the i-th replica R receives a PROPOSE message m from P, it checks whether at least **nf** other replicas received the same proposal m from primary P. This check assures R that at least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas received the same proposal, which will play a central role in achieving speculative non-divergence. To perform this check, each replica *supports* the first proposal m it receives from the primary by computing a *signature share* $s\langle m \rangle_i$ and sending a SUPPORT message containing this share to the primary.

The primary P waits for SUPPORT messages with valid signature shares from **nf** distinct replicas, which can then be aggregated into a single signature $\langle m \rangle$. After generating such a signature, the primary broadcasts this signature to all replicas via a CERTIFY message.

After a replica R receives a valid CERTIFY message, it *view-commits* to T as the k-th transaction in view v. The replica logs this view-commit decision as $\texttt{VCommit}_R(\langle T \rangle_c, v, k)$. After R view-commits to T, R schedules T for speculative execution as the k-th transaction of view v. Consequently, T will be executed by R after all preceding transactions are executed. We write $\texttt{Execute}_R(\langle T \rangle_c, v, k)$ to log this execution.

After execution, R informs the client of the order of execution and of execution result r (if any) via a message INFORM. In turn, client c will wait for a *proof-of-execution* for the transaction T it requested, which consists of identical INFORM messages from **nf** distinct replicas. This proof-of-execution guarantees that at least $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas executed T as the k-th transaction and in Section 3.3, we will see that such transactions are always preserved by PoE when recovering from failures.

If client c does not know the current primary or does not get any timely response for its requests, then it can broadcast its request $\langle T \rangle_c$ to all replicas. The non-faulty replicas will then forward this request to the current primary (if T is not yet executed) and ensure that the primary initiates successful proposal of this request in a timely manner.

To prove correctness of PoE in all cases, we will need the following technical safety-related property of view-commits.

Client-role (used by client c to request transaction T) :

1: Send $\langle T \rangle_c$ to the primary P.
2: Await receipt of messages INFORM$(\langle T \rangle_c, v, k, r)$ from **nf** replicas.
3: Considers T executed, with result r, as the k-th transaction.

Primary-role (running at the primary P of view v, $\text{id}(\text{P}) = v \bmod \mathbf{n}$) :

4: Let view v start after execution of the k-th transaction.
5: **event** P awaits receipt of message $\langle T \rangle_c$ from client c **do**
6: Broadcast PROPOSE$(\langle T \rangle_c, v, k)$ to all replicas.
7: $k := k + 1$.
8: **end event**
9: **event** P receives **nf** message SUPPORT$(s\langle h \rangle_i, v, k)$ such that:
(1) each message was sent by a distinct replica, $i \in \{1, \ldots, n\}$; and
(2) All $s\langle h \rangle_i$ in this set can be combined to generate signature $\langle h \rangle$.
do
10: Broadcast CERTIFY$(\langle h \rangle, v, k)$ to all replicas.
11: **end event**

Backup-role (running at every i-th replica R.) :

12: **event** R receives message $m :=$ PROPOSE$(\langle T \rangle_c, v, k)$ such that:
(1) v is the current view;
(2) m is sent by the primary of v; and
(3) R did not accept a k-th proposal in v
do
13: Compute $h := \text{D}(\langle T \rangle_c || v || k)$.
14: Compute signature share $s\langle h \rangle_i$.
15: Transmit SUPPORT$(s\langle h \rangle_i, v, k)$ to P.
16: **end event**
17: **event** R receives messages CERTIFY$(\langle h \rangle, v, k)$ from P such that:
(1) R transmitted SUPPORT$(s\langle h \rangle_i, v, k)$ to P; and
(2) $\langle h \rangle$ is a valid threshold signature
do
18: View-commit T, the k-th transaction of v ($\texttt{VCommit}_R(\langle T \rangle_c, v, k)$).
19: **end event**
20: **event** R logged $\texttt{VCommit}_R(\langle T \rangle_c, v, k)$ and
has logged $\texttt{Execute}_R(t', v', k')$ for all $0 \leq k' < k$ **do**
21: Execute T as the k-th transaction of v ($\texttt{Execute}_R(\langle T \rangle_c, v, k)$).
22: Let r be the result of execution of T (if there is any result).
23: Send INFORM$(\text{D}(\langle T \rangle_c), v, k, r)$ to c.
24: **end event**

Figure 3: The normal-case algorithm of PoE.

PROPOSITION 3.2. *Let* R_i, $i \in \{1, 2\}$, *be two non-faulty replicas that view-committed to* $\langle T_i \rangle_{c_i}$ *as the* k*-th transaction of view* v ($\texttt{VCommit}_R(\langle T \rangle_c, v, k)$). *If* $\mathbf{n} > 3\mathbf{f}$, *then* $\langle T_1 \rangle_{c_1} = \langle T_2 \rangle_{c_2}$.

PROOF. Replica R_i only view-committed to $\langle T_i \rangle_{c_i}$ after R_i received CERTIFY$(\langle h \rangle, v, k)$ from the primary P (Line 17 of Figure 3). This message includes a threshold signature $\langle h \rangle$, whose construction requires signature shares from a set S_i of **nf** distinct replicas. Let $X_i = S_i \setminus \mathcal{F}$ be the non-faulty replicas in S_i. As $|S_i| = \mathbf{nf}$ and $|\mathcal{F}| = \mathbf{f}$, we have $|X_i| \geq \mathbf{nf} - \mathbf{f}$. The non-faulty replicas in X_i will only send a single SUPPORT message for the k-th transaction in view v (Line 12 of Figure 3). Hence, if $\langle T_1 \rangle_{c_1} \neq \langle T_2 \rangle_{c_2}$, then X_1 and X_2 must not overlap and $\mathbf{nf} \geq |X_1 \cup X_2| \geq 2(\mathbf{nf} - \mathbf{f})$ must hold. As $\mathbf{n} = \mathbf{nf} + \mathbf{f}$, this simplifies to $3\mathbf{f} \geq \mathbf{n}$, which contradicts $\mathbf{n} > 3\mathbf{f}$. Hence, we conclude $\langle T_1 \rangle_{c_1} = \langle T_2 \rangle_{c_2}$. □

We will later use Proposition 3.2 to show that PoE provides speculative non-divergence. Next, we look at typical cases in which the normal-case of PoE is interrupted:

Example 3.3. A malicious primary can try to affect PoE by not conforming to the normal-case algorithm in the following ways:

(1) By sending proposals for different transactions to different non-faulty replicas. In this case, Proposition 3.2 guarantees that at most a single such proposed transaction will get view-committed by any non-faulty replica.
(2) By keeping some non-faulty replicas in the dark by not sending proposals to them. In this case, the remaining non-faulty replicas can still end up view-committing the transactions as long as at least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas receive proposals: the faulty replicas in $\mathcal{F}$ can take over the

role of up to **f** non-faulty replicas left in the dark (giving the false illusion that the non-faulty replicas in the dark are malicious).

(3) By preventing execution by not proposing a k-th transaction, even though transactions following the k-th transaction are being proposed.

When the network is unreliable and messages do not get delivered (or not on time), then the behavior of a non-faulty primary can match that of the malicious primary in the above example. Indeed, failure of the normal-case of PoE has only two possible causes: primary failure and unreliable communication. If communication is unreliable, then there is no way to guarantee continuous service [19]. Hence, replicas simply assume failure of the current primary if the normal-case behavior of PoE is interrupted, while the design of PoE guarantees that unreliable communication does not affect the correctness of PoE.

To deal with primary failure, each replica maintains a timer for each request. If this timer expires (*timeout*) and it has not been able to execute the request, it assumes that the primary is malicious. To deal with such a failure, replicas will replace the primary. Next, we present the *view-change algorithm* that performs primary replacement.

3.3 The View-Change Algorithm

If PoE observes failure of the primary P of view v, then PoE will elect a new primary and move to the next view, view $v+1$, via the *view-change algorithm*. The goals of the view-change are

(1) to assure that each request that *is considered executed* by any client is preserved under all circumstances; and

(2) to assure that the replicas are able to agree on a new view whenever communication is reliable.

As described in the previous section, a client will consider its request executed if it receives a *proof-of-execution* consisting of identical INFORM responses from at-least **nf** distinct replicas. Of these **nf** responses, at-most **f** can come from faulty replicas. Hence, a client can only consider its request executed whenever the requested transaction was executed (and view-committed) by at-least $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas in the system. We note the similarity with the view-change algorithm of PBFT, which will preserve any request that is *prepared* by at-least $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas.

The view-change algorithm of PoE consists of three steps. First, failure of the current primary P needs to be detected by all non-faulty replicas. Second, all replicas exchange information to establish which transactions were included in view v and which were not. Third, the new primary P′ proposes a new view. This new view proposal contains a list of the transactions executed in the previous views (based on the information exchanged earlier). Finally, if the new view proposal is valid, then replicas switch to this view; otherwise, replicas detect failure of P′ and initiate a view-change for the next view ($v+2$). The communication of the view-change algorithm of PoE is sketched in Figure 4 and the full pseudo-code of the algorithm can be found in Figure 5. Next, we discuss each step in detail.

3.3.1 Failure Detection and View-Change Requests. If a replica R detects failure of the primary of view v, then it halts the normal-case algorithm of PoE for view v and informs all other replicas of this failure by requesting a view-change. The replica R does so by broadcasting a message VC-REQUEST(v, E), in which E is a summary of all transactions executed by R (Figure 5, Line 1). Each replica R can detect the failure of primary in two ways:

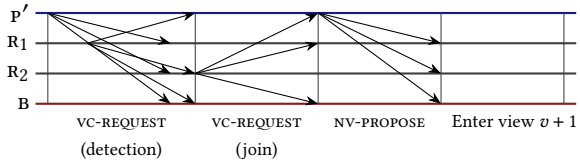


Figure 4: The current primary B of view v is faulty and needs to be replaced. The next primary, P′, and the replica R_1 detected this failure first and request view-change via VC-REQUEST messages. The replica R_2 joins these requests.

vc-request (used by replica R to request view-change) **:**

1: **event** R detects failure of the primary **do**
2: R halts the normal-case algorithm of Figure 3 for view v.
3: $E := \{(\text{CERTIFY}(\langle h \rangle, w, k), \langle T \rangle_c) \mid w \leq v$ and $\text{Execute}_R(\langle T \rangle_c, w, k)$ and $h = \text{D}(\langle T \rangle_c || w || k)\}$.
4: Broadcast VC-REQUEST(v, E) to all replicas.
5: **end event**
6: **event** R receives **f** + 1 messages VC-REQUEST(v_i, E_i) such that
(1) each message was sent by a distinct replica; and
(2) v_i, $1 \leq i \leq \mathbf{f}+1$, is the current view
do
7: R detects failure of the primary (join).
8: **end event**

On receiving nv-propose (use by replica R) **:**

9: **event** R receives $m =$ NV-PROPOSE$(v+1, m_1, m_2, ..., m_{\mathbf{nf}})$ **do**
10: **if** m is a valid new-view proposal (similar to creating NV-PROPOSE) **then**
11: Derive the transactions N for the new-view from $m_1, m_2, \ldots, m_{\mathbf{nf}}$.
12: Rollback any executed transactions not included in N.
13: Execute the transactions in N not yet executed.
14: Move into view $v+1$ (see Section 3.3.3 for details).
15: **end if**
16: **end event**

nv-propose (used by replica P′ that will act as the new primary) **:**

17: **event** P′ receives **nf** messages $m_i =$ VC-REQUEST(v_i, E_i) such that
(1) these messages are sent by a set S, $|S| = \mathbf{nf}$, of distinct replicas;
(2) for each m_i, $1 \leq i \leq \mathbf{nf}$, sent by replica $Q_i \in S$, E_i consists of a consecutive sequence of entries (CERTIFY$(\langle h \rangle, v, k)$, $\langle T \rangle_c$);
(3) v_i, $1 \leq i \leq \mathbf{nf}$, is the current view v; and
(4) P′ is the next primary (id(P′) $= (v+1) \bmod \mathbf{n}$)
do
18: Broadcast NV-PROPOSE$(v+1, m_1, m_2, ..., m_{\mathbf{nf}})$ to all replicas.
19: **end event**

Figure 5: The view-change algorithm of PoE.

(1) R *timeouts* while expecting normal-case operations toward executing a client request. E.g., when R forwards a client request to the current primary, and the current primary fails to propose this request on time.

(2) R receives VC-REQUEST messages, indicating that the primary of view v failed, from **f** + 1 distinct replicas. As at most **f** of these messages can come from faulty replicas, at least one non-faulty replica must have detected a failure. In this case, R joins the view-change (Figure 5, Line 6).

3.3.2 Proposing the New View. To start view $v+1$, the new primary P′ (with id(P′) $= (v+1) \bmod \mathbf{n}$) needs to propose a new view by determining a valid list of requests that need to be preserved. To do so, P′ waits until it receives sufficient information. In specific, P′ waits until it received *valid* VC-REQUEST messages from a set $S \subseteq \mathfrak{R}$ of $|S| = \mathbf{nf}$ distinct replicas.

An i-th view-change request m_i is considered valid if it includes a *consecutive sequence* of pairs $(c, \langle T \rangle_c)$, where c is a valid CERTIFY message for request $\langle T \rangle_c$. Such a set S is guaranteed to exist when communication is reliable, as all non-faulty replicas will participate in the view-change algorithm. The new primary

collects the set S of $|S| = \mathbf{nf}$ valid VC-REQUEST and proposes them in a new view message NV-PROPOSE to all replicas.

3.3.3 Move to the New View. After a replica R receives a NV-PROPOSE message containing a new-view proposal from the new primary P′, R validates the content of this message. From the set of VC-REQUEST messages in the new-view proposal, R chooses, for each k, the pair $(\text{CERTIFY}(\langle h \rangle, w, k), \langle T \rangle_c)$ proposed in the most-recent view w. Furthermore, R determines the total number of such requests $k_{\max}$. Then, R view-commits and executes all $k_{\max}$ chosen requests that happened before view $v + 1$. Notice that replica R can skip execution of any transaction it already executed. If R executed transactions not included in the new-view proposal, then R needs to *rollback* these transactions before it can proceed executing requests in view $v + 1$. After these steps, R can switch to the new view $v + 1$. In the new view, the new primary P′ starts by proposing the $k_{\max} + 1$-th transaction.

When moving into the new view, we see the cost of speculative execution: some replicas can be forced to *rollback execution* of transactions:

Example 3.4. Consider a system with non-faulty replica R. When deciding the k-th request, communication became unreliable, due to which only R received a CERTIFY message for request $\langle T \rangle_c$. Consequently, R speculatively executes T and informs the client c. During the view-change, all other replicas—none of which have a CERTIFY message for $\langle T \rangle_c$—provide their local state to the new primary, which proposes a new view that does not include any k-th request. Hence, the new primary will start its view by proposing client request $\langle T' \rangle_{c'}$ as the k-th request, which gets accepted. Consequently, R needs to rollback execution of T. Luckily, this is not an issue: the client c only got at-most $\mathbf{f}+1 < \mathbf{nf}$ responses for request, does not yet have a proof-of-execution, and, consequently, does not consider T executed.

In practice, rollbacks can be supported by, e.g., undoing the operations of transaction in reverse order, or by reverting to an old state. For the correct working of PoE, the exact working of rollbacks is not important as long as the execution layer provides support for rollbacks.

3.4 Correctness of PoE

First, we show that the normal-case algorithm of PoE provides non-divergent speculative consensus when the primary is non-faulty and communication is reliable.

THEOREM 3.5. *Consider a system in view v, in which the first $k-1$ transactions have been executed by all non-faulty replicas, in which the primary is non-faulty, and communication is reliable. If the primary received $\langle T \rangle_c$, then the primary can use the algorithm in Figure 3 to ensure that*

(1) there is non-divergent execution of T;
(2) c considers T executed as the k-th transaction; and
(3) c learns the result of executing T (if any),

this independent of any malicious behavior by faulty replicas.

PROOF. Each non-faulty primary would follow the algorithm of PoE described in Figure 3 and send $\text{PROPOSE}(\langle T \rangle_c, v, k)$ to all replicas (Line 6). In response, all **nf** non-faulty replicas will compute a signature share and send a SUPPORT message to the primary (Line 15). Consequently, the primary will receive signature shares from **nf** replicas and will combine them to generate a threshold signature $\langle h \rangle$. The primary will include this signature $\langle h \rangle$ in a CERTIFY message and broadcast it to all replicas. Each replica will successfully verify $\langle h \rangle$ and will view-commit to T (Line 17). As the first $k-1$ transactions have already been executed, every non-faulty replica will execute T. As all non-faulty replicas behave deterministically, execution will yield the same result r (if any) across all non-faulty replicas. Hence, when the non-faulty replicas inform c, they do so by all sending identical messages $\text{INFORM}(\text{D}(\langle T \rangle_c), v, k, r)$ to c (Line 20–Line 23). As all **nf** non-faulty replicas executed T, we have non-divergent execution. Finally, as there are at most **f** faulty replicas, the faulty replicas can only forge up to **f** invalid INFORM messages. Consequently, the client c will only receive the message $\text{INFORM}(\text{D}(\langle T \rangle_c), v, k, r)$ from at least **nf** distinct replicas, and will conclude that T is executed yielding result r (Line 3). □

At the core of the correctness of PoE, under all conditions, is that no replica will rollback requests $\langle T \rangle_c$ for which client c already received a proof-of-execution. We prove this next:

PROPOSITION 3.6. *Let $\langle T \rangle_c$ be a request for which client c already received a proof-of-execution showing that T was executed as the k-th transaction of view v. If $\mathbf{n} > 3\mathbf{f}$, then every non-faulty replica that switches to a view $v' > v$ will preserve T as the k-th transaction of view v.*

PROOF. Client c considers $\langle T \rangle_c$ executed as the k-th transaction of view v when it received identical INFORM-messages for T from a set A of $|A| = \mathbf{nf}$ distinct replicas (Figure 3, Line 3). Let $B = A \setminus \mathcal{F}$ be the set of non-faulty replicas in A.

Now consider a non-faulty replica R that switches to view $v' > v$. Before doing so, R must have received a valid proposal $m = \text{NV-PROPOSE}(v', m_1, ..., m_{\mathbf{nf}})$ from the primary of view v'. Let C be the set of **nf** distinct replicas that provided messages $m_1, \ldots, m_{\mathbf{nf}}$ and let $D = C \setminus \mathcal{F}$ be the set of non-faulty replicas in C. We have $|B| \geq \mathbf{nf} - \mathbf{f}$ and $|D| \geq \mathbf{nf} - \mathbf{f}$. Hence, using a contradiction argument similar to the one in the proof of Proposition 3.2, we conclude that there must exists a non-faulty replica $\text{Q} \in (B \cap D)$ that executed $\langle T \rangle_c$, informed c, and requested a view-change.

To complete the proof, we need to show that $\langle T \rangle_c$ was proposed and executed in the last view that proposed and view-committed a k-th transaction and, hence, that Q will include $\langle T \rangle_c$ in its VC-REQUEST message for view v'. We do so by induction on the difference $v' - v$. As the base case, we have $v' - v = 1$, in which case no view after v exists yet and, hence, $\langle T \rangle_c$ must be the newest k-th transaction available to Q. As the induction hypothesis, we assume that all non-faulty replicas will preserve T when entering a new view w, $v < w \leq w'$. Hence, non-faulty replicas participating in view w will not support any k-th transactions proposed in view w. Consequently, no CERTIFY messages can be constructed for any k-th transaction in view w. Hence, the new-view proposal for $w' + 1$ will include $\langle T \rangle_c$, completing the proof. □

As a direct consequence of the above, we have

COROLLARY 3.7 (SAFETY OF POE). *PoE provides speculative non-divergence if $\mathbf{n} > 3\mathbf{f}$.*

We notice that the view-change algorithm does not deal with minor malicious behavior (e.g., a single replica left in the dark). Furthermore, the presented view-change algorithm will recover all transactions since the start of the system, which will result in unreasonable large messages when many transactions have already been proposed. In practice, both these issues can be resolved by regularly making *checkpoints* (e.g., after every 100 requests) and only including requests since the last checkpoint in each VC-REQUEST message. To do so, PoE uses a standard fully-decentralized PBFT-style checkpoint algorithm that enables the independent checkpointing and recovery of any request that is

executed by at least $\mathbf{f}+1$ non-faulty replicas whenever communication is reliable [9]. Finally, utilizing the view-change algorithm and checkpoints, we prove

Theorem 3.8 (Liveness of PoE). *PoE provides termination in periods of reliable bounded-delay communication if* $\mathbf{n} > 3\mathbf{f}$.

Proof. When the primary is non-faulty, Theorem 3.5 guarantees termination as replicas continuously accept and execute requests. If the primary is Byzantine and fails to guarantee termination for at most $\mathbf{f}$ non-faulty replicas, then the checkpoint algorithm will assure termination of these non-faulty replicas. Finally, if the primary is Byzantine and fails to guarantee termination for at least $\mathbf{f}+1$ non-faulty replicas, then it will be replaced using the view-change algorithm. For the view-change process, each replica will start with a timeout δ after it receives $\mathbf{nf}$ matching vc-requests and double this timeout after each view-change (exponential backoff). When communication becomes reliable, this mechanism guarantees that all replicas will eventually view-change to the same view at the same time. After this point, a non-faulty replica will become primary in at most $\mathbf{f}$ view-changes, after which Theorem 3.5 guarantees termination. □

3.5 Fine-Tuning and Optimizations

To keep presentation simple, we did not include the following optimizations in the protocol description:

(1) To reach $\mathbf{nf}$ signature shares, the primary can generate one itself. Hence, it only needs $\mathbf{nf}-1$ shares of other replicas.
(2) The propose, support, inform, and nv-propose messages are not forwarded and only need MACs to provide message authentication. The certify messages need not be signed, as tampering them would invalidate the threshold signature. The vc-request messages need to be signed, as they need to be forwarded without tampering.

Finally, the design of PoE is fully compatible with *out-of-order processing* as a replica only supports proposals for a k-th transaction if it had not previously supported another k-th proposal (Figure 3, Line 12) and only executes a k-th transaction if it has already executed all the preceding transactions (Figure 3, Line 20). As the size of the active out-of-order processing window determines how many client requests are being processed at the same time (without receiving a proof-of-execution), the size of the active window determines the number of transactions that can be rolled back during view-changes.

3.6 Designing PoE using MACs

The design of PoE can be adapted to only use message authentication codes (MACs) to authenticate communication. This will sharply reduce the computational complexity of PoE and eliminate one round of communication, this at the cost of higher *quadratic* overall communication costs (see Figure 2).

The usage of only MACs makes it impossible to obtain threshold signatures or reliably forward messages (as forwarding replicas can tamper with the content of unsigned messages). Hence, using MACs requires changes to how client requests are included in proposals (as client requests are forwarded), to the normal-case algorithm of PoE (which uses threshold signatures), and to the view-change algorithm of PoE (which forwards vc-request messages). The changes to the proposal of client requests and to the view-change algorithm can be derived from the strategies used by Pbft to support MACs [9]. Hence, next we only review the changes to the normal-case algorithm of PoE.

Consider a replica r that receives a propose message from the primary p. Next, r needs to determine whether at least $\mathbf{nf}$ other replicas received the same proposal, which is required to achieve speculative non-divergence (see Proposition 3.2). When using MACs, r can do so by replacing the all-to-one support and one-to-all certify phases by a single all-to-all *support phase*. In the support phase, each replica agrees to *support* the first proposal $\text{propose}(\langle T \rangle_c, v, k)$ it receives from the primary by broadcasting a message $\text{support}(\text{D}(\langle T \rangle_c), v, k)$ to all replicas. After this broadcast, each replica waits until it receives support messages, identical to the message it sent, from $\mathbf{nf}$ distinct replicas. If r receives these messages, it *view-commits* to T as the k-th transaction in view v and schedules T for execution. We have sketched this algorithm in Figure 2.

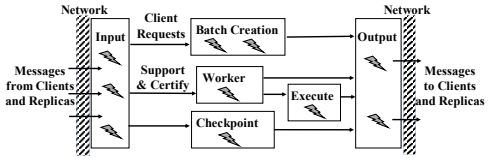


Figure 6: Multi-threaded Pipelines at different replicas.

4 RESILIENTDB FABRIC

To test our design principles in practical settings, we implement our PoE protocol in our ResilientDB fabric [27–31]. ResilientDB provides its users access to a state-of-the-art replicated transactional engine and fulfills the need of a high-throughput permissioned blockchain fabric. ResilientDB helps us to realize the following goals: (i) implement and test different consensus protocols; (ii) balance the tasks done by a replica through a *parallel pipelined architecture*; (iii) minimize the cost of communication through *batching* client transactions; and (iv) enable use of a secure and efficient ledger. Next, we present a brief overview of our ResilientDB fabric.

ResilientDB lays down a *client-server* architecture where clients send their transactions to servers for processing. We use Figure 6 to illustrate the multi-threaded pipelined architecture associated with each replica. At each replica, we spawn multiple *input* and *output* threads for communicating with the network.

Batching. During our formal description of PoE, we assumed that the propose message from the primary includes a single client request. An effective way to reduce the overall cost of consensus is by aggregating several client requests in a single batch and use one consensus step to reach agreement on all these requests [9, 21, 38]. To maximize performance, ResilientDB facilitates batching requests at both replicas and clients.

At the primary replica, we spawn multiple *batch-threads* that aggregate clients requests into a batch. The input-threads at the primary receive client requests, assign them a sequence number and enqueue these requests in the *batch-queue*. In ResilientDB, all batch-threads share a common *lock-free queue*. When a client request is available, a batch-thread dequeues the request and continues adding it to an existing batch until the batch has reached a pre-defined size. Each batching-thread also hashes the requests in a batch to create a unique digest.

All other messages received at a replica are enqueued by the input-thread in the *work-queue* to be processed by the single *worker-thread*. Once a replica receive a certify message from the primary, it forwards the request to the *execute-thread* for execution. Once the execution is complete, the execution-thread creates an inform message, which is transmitted to the client.

Ledger Management. We now explain how we efficiently maintain a blockchain ledger across different replicas. A blockchain is an immutable ledger, where blocks are chained as a linked-list. An i-th block can be represented as $B_i := \{k, d, v,$

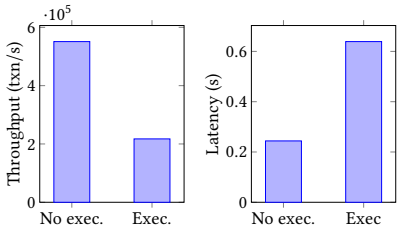


Figure 7: Upper bound on performance when primary only replies to clients (*No exec.*) and when primary executes a request and replies to clients (*Exec.*).

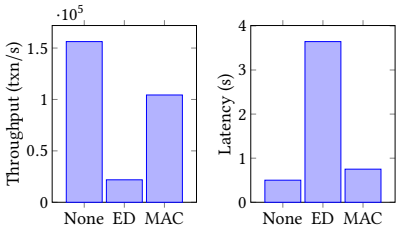


Figure 8: System performance using three different signature schemes. In all cases, n = 16 replicas participate in consensus.

$H(B_{i-1})\}$, in which k is the sequence number of the client request, d the digest of the request, v the view number, and $H(B_{i-1})$ the hash of the previous block. In ResilientDB, prior to any consensus, we require the *first* primary replica to create a *genesis block* [31]. This genesis block acts as the first block in the blockchain and contains some basic data. We use the hash of the identity of the initial primary, as this information is available to each participating replicas (eliminating the need for any extra communication to exchange this block).

After the genesis block, each replica can independently create the next block in the blockchain. As stated above, each block corresponds to some batch of transactions. A block is only created by the execute-thread once it completes executing a batch of transactions. To create a block, the execute-thread hashes the previous block in the blockchain and creates a *new block*. To prove the validity of individual blocks, ResilientDB stores the *proof-of-accepting the k-th request* in the k-th block. In PoE, such a proof includes the threshold signature sent by the primary as part of the certify message.

5 EVALUATION

We now analyze our design principles in practice. To do so, we evaluate our PoE protocol against four state-of-the-art bft protocols. There are many bft protocols we could compare with. Hence, we pick a representative sample: (1) Zyzzyva—as it has the absolute minimal cost in the fault-free case, (2) Pbft—as it is a common baseline (the used design is based on BFTSmart [7]), (3) SBFT—as it is a safer variation of Zyzzyva, and (3) HotStuff—as it is a linear-communication protocol that adopts the notion of rotating leaders. Through our experiments, we want to answer the following questions:

(Q1) How does PoE fare in comparison with the other protocols under failures?
(Q2) Does PoE benefits from batching client requests?
(Q3) How does PoE perform under zero payload?
(Q4) How scalable is PoE on increasing the number of replicas participating in the consensus, in the normal-case?

Setup. We run our experiments on the Google Cloud, and deploy each replicas on a *c2* machine having a 16-core Intel Xeon Cascade Lake CPU running at 3.8 GHz with 32 GB memory. We deploy up to 320 k clients on 16 machines. To collect results after reaching a steady-state, we run each experiment for 180 s: the first 60 s are warmup, and measurement results are collected over the next 120 s. We average our results over three runs.

Configuration and Benchmarking. For evaluating the protocols, we employed YCSB [13] from Blockbench's macro benchmarks [16]. Each client request queries a YCSB table that holds half a million active records. We require 90% of the requests to be write queries as the majority of typical blockchain transactions are updates to existing records. Prior to the experiments, each replica is initialized with an identical copy of the YCSB table. The client requests generated by YCSB follow a Zipfian distribution and are heavily skewed (skew factor 0.9).

Unless *explicitly* stated, we use the following configuration for all experiments. We perform scaling experiments by varying replicas from 4 to 91. We divide our experiments in two dimensions: (1) *Zero Payload* or *Standard Payload*, and (2) *Failures* or *Non-Failures*. We employ batching with a batch size of 100 as the percentage increase in throughput on larger batch sizes is small.

Under Zero Payload conditions, all replicas execute 100 dummy instructions per batch, while the primary sends an empty proposal (and not a batch of 100 requests). Under Standard Payload, with a batch size of 100, the size of Propose message is 5400 B, of Response message is 1748 B, and of other messages is around 250 B. For experiments with failures, we force one backup replica to crash. Additionally, we present an experiment that illustrates the effect of primary failure. We measure *throughput* as transactions executed per second. We measure *latency* as the time from when the client sends a request to the time when the client receives a response.

Other protocols: We also implement Pbft, Zyzzyva, SBFT and HotStuff in our ResilientDB fabric. We refer to Section 2 for further details on the working of Zyzzyva, SBFT, and HotStuff. Our implementation of Pbft is based on the BFTSmart [7] framework with the added benefits of out-of-order processing, pipelining, and multi-threading. In both Pbft and Zyzzyva, digital signatures are used for authenticating messages sent by the clients, while MACs are used for other messages. Both SBFT and HotStuff require threshold signatures for their communication.

5.1 System Characterization

We first determine the upper bounds on the performance of ResilientDB. In Figure 7, we present the maximum throughput and latency of ResilientDB when there is *no communication* among the replicas. We use the term *No Execution* to refer to the case where all clients send their request to the primary replica and primary simply responds back to the client. We count every query responded back in the system throughput. We use the term *Execution* to refer to the case where the primary replica executes each query before responding back to the client.

The architecture of ResilientDB (see Section 4) states the use of one worker thread. In these experiments, we maximize system performance by allowing up to two threads to work independently at the primary replica without ordering any queries. Our results indicate that the system can attain high throughputs (up to 500 ktxn/s) and can reach low latencies (up to 0.25 s). Notice that if we employ additional worker-threads, our ResilientDB fabric can easily attain higher throughput.

5.2 Effect of Cryptographic Signatures.

ResilientDB enables a flexible design where replicas and clients can employ both digital signatures (threshold signatures) and

message authentication codes. This helps us to implement PoE and other consensus protocols in ResilientDB.

To achieve authenticated communication using symmetric cryptography, we employ a combination of CMAC and AES [36]. Further, we employ ED25519-based digital signatures to enable asymmetric cryptographic signing. For generating efficient threshold signature scheme, we use Boneh–Lynn–Shacham (BLS) signatures [36]. To create message digests and for hashing purposes, we use the SHA256 algorithm.

Next, we determine the cost of different cryptographic signing schemes. For this purpose, we run three different experiments in which (i) no signature scheme is used (*None*); (ii) everyone uses digital signatures based on ED25519 (*ED*); and (iii) all replicas use CMAC+AES for signing, while clients sign their message using ED25519 (*MAC*). In these three experiments, we run Pbft consensus among 16 replicas. In Figure 8, we illustrate the throughput attained and latency incurred by ResilientDB for the experiments. Clearly, the system attains its highest throughput when no signatures are employed. However, such a system cannot handle malicious attacks. Further, using just digital signatures for signing messages can prove to be expensive. An optimal configuration can require clients to sign their messages using digital signatures, while replicas can communicate using MACs.

5.3 Scaling Replicas under Standard Payload

In this section, we evaluate scalability of PoE both under backup failure and no failures.

(1) ***Single Backup Failure.*** We use Figures 9(a) and 9(b) to illustrate the throughput and latency attained by the system on running different consensus protocols under a backup failure. These graphs affirm our claim that PoE attains higher throughput and incurs lower latency than all other protocols.

In case of Pbft, each replica participates in two phases of quadratic communication, which limits its throughput. For the twin-path protocols such as Zyzzyva and SBFT, a single failure is sufficient to cause massive reductions in their system throughputs. Notice that the collector in SBFT and the clients in Zyzzyva have to wait for messages from all **n** replicas, respectively. As predicting an optimal value for timeouts is hard [11, 12], we chose a very small value for the timeout (3 s) for replicas and clients. We justify these values, as the experiments we show later in this section show that the average latency can be as large as 6 s. We note that high timeouts affect Zyzzyva more than SBFT. In Zyzzyva, clients are waiting for timeouts during which they stop sending requests, which empties the pipeline at the primary, starving it from new request to propose. To alleviate such issues in real-world deployments of Zyzzyva, clients need to be able to precisely predict the latency to minimize the time the clients needs to wait between requests. Unfortunately, this is hard and runs the risk of ending up in the expensive slow path of Zyzzyva whenever the predicted latency is slightly off. In SBFT, the collector may timeout waiting for threshold shares for the k-th round while the primary can continues propose requests for future round l, $l > k$. Hence, in SBFT replicas have more opportunity to occupy themselves with useful work.

HotStuff attains significantly low throughput due to its sequential primary-rotation model in which each of its primaries has to wait for the previous primary before proposing the next request, which leads to a huge reduction in its throughput. Interestingly, HotStuff incurs the least average latency among all protocols. This is a result of intensive load on the system when running other protocols. As these protocols process several requests concurrently (see the multi-threaded architecture in Section 4), these requests spend on average more time in the queue before being processed by a replica. Notice that all out-of-order consensus protocols employ this trade off: a small sacrifice on latency yields higher gains on system throughput.

In case of PoE, its high throughputs under failures is a result of its three-phase linear protocol that does not rely on any twin-path model. To summarize, PoE attains up to 43%, 72%, 24× and 62× more throughputs than Pbft, SBFT, HotStuff and Zyzzyva.

(2) ***No Replica Failure.*** We use Figures 9(c) and 9(d) to illustrate the throughput and latency attained by the system on running different consensus protocols in fault-free conditions. These plots help us to bound the maximum throughput that can be attained by different consensus protocols in our system.

First, as expected, in comparison to the Figures 9(a) and 9(b), the throughputs for PoE and Pbft are slightly higher. Second, PoE continues to outperform both Pbft and HotStuff, for the reasons described earlier. Third, both Zyzzyva and SBFT are now attaining higher throughputs as their clients and collector no longer timeout, respectively. The key reason SBFT's gains are limited is because SBFT requires five phases and becomes computation bounded. Although Pbft is quadratic, it employs MAC, which are cheaper to sign and verify.

Notice that the differences in throughputs of PoE and Zyzzyva are small. PoE has 20% (on 91 replicas) to 13% (on 4 replicas) less throughputs than Zyzzyva. An interesting observation is that on 91 replicas, Zyzzyva incurs almost the same latency as PoE, even though it has higher throughput. This happens as clients in PoE have to wait for only the fastest $\mathbf{nf} = 61$ replies, whereas a client for Zyzzyva has to wait for replies from all replicas (even the slowest ones). To conclude, PoE attains up to 35%, 27% and 21× more throughput than Pbft, SBFT and HotStuff, respectively.

5.4 Scaling Replicas under Zero Payload

We now measure the performance of different protocols under zero payload. In any bft protocol, the primary starts consensus by sending a Propose message that includes all transactions. As a result, this message has the largest size and is responsible for consuming the majority of the bandwidth. A zero payload experiment ensures that each replica executes dummy instructions. Hence, the primary is no longer a bottleneck.

We again run these experiments for both ***Single Failure*** and ***Failure-Free*** cases, and use Figures 9(e) to 9(h) to illustrate our observations. It is evident from these figures that zero payload experiments have helped in increasing PoE's gains. PoE attains up to 85%, 62% and 27× more throughputs than Pbft, SBFT and HotStuff, respectively. In fact, under failure-free conditions, the throughput attained by PoE is comparable to Zyzzyva. This is easily explained. First, both PoE and Zyzzyva are linear protocols. Second, although in failure-free cases Zyzzyva attains consensus in one phase, its clients need to wait for response from all **n** replicas, which gives PoE an opportunity to cover the gap. However, SBFT being a linear protocol does not perform as good as its other linear counterparts. Its throughput is impacted by the delay of five phases.

5.5 Impact of Batching under Failures

Next, we study the effect of batching client requests on bft protocols [9, 51]. To answer this question, we measure performance as function of the number of requests in a batch (*the batch-size*), which we vary between 10 and 400. For this experiment, we use a system with 32 available replicas, of which one replica has failed.

We use Figures 9(i) and 9(j) to illustrate, for each consensus protocol, the throughput and average latency attained by the system. For each protocol, increasing the batch-size also increases

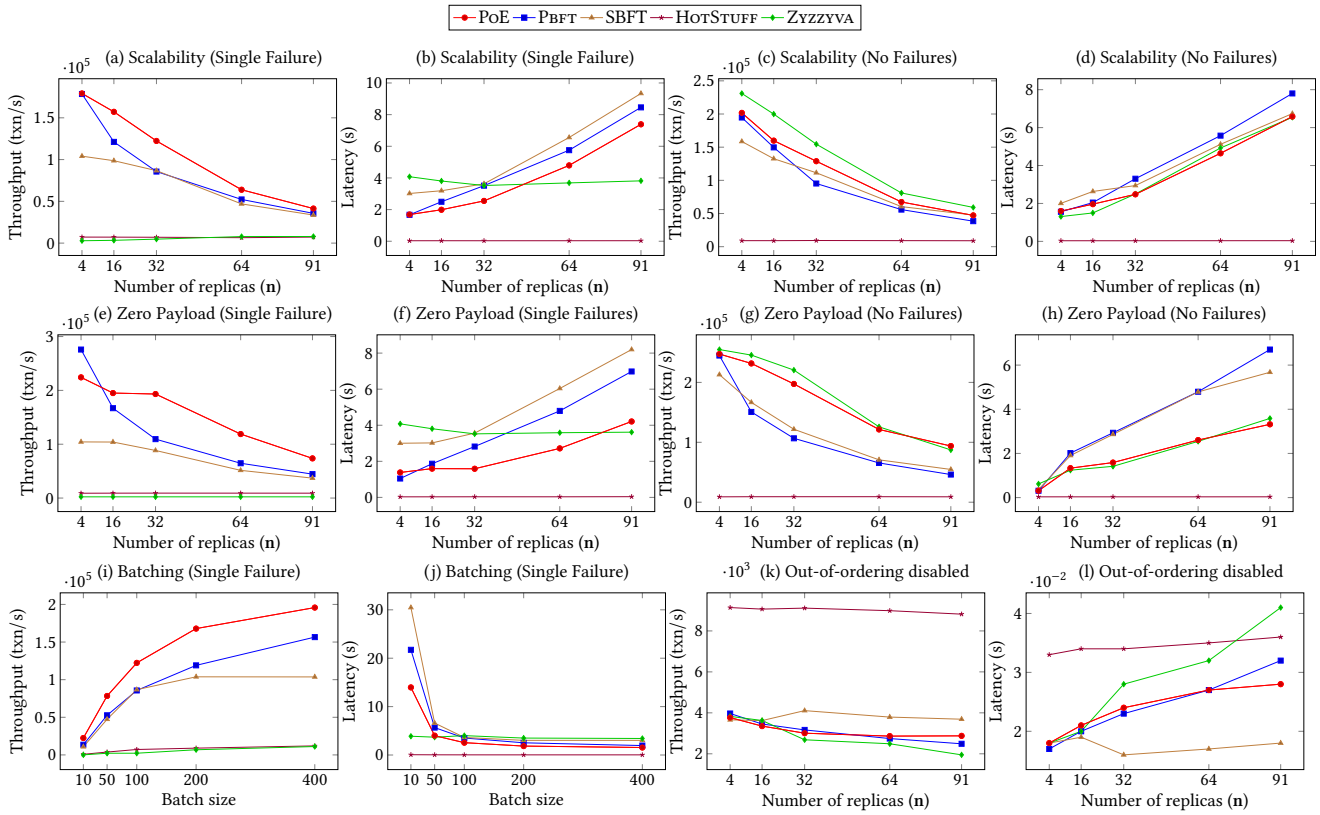


Figure 9: Evaluating system throughput and average latency incurred by PoE and other BFT protocols.

throughput, while decreasing the latency. This happens as larger batch-sizes require fewer consensus rounds to complete the exact same set of requests, reducing the cost of ordering and executing the transactions. This not only improves throughput, but also reduces client latencies as clients receive faster responses for their requests. Although increasing the batch-size reduces the number of consensus rounds, the large message size causes a proportional decrease in throughput (or increase in latency). This is evident from the experiments at higher batch-sizes: increasing the batch-size beyond 100 gradually curves the throughput plots towards a limit for PoE, Pbft and SBFT. For example, on increasing the batch size from 100 to 400, PoE and Pbft see an increase in throughput by 60% and 80%, respectively, while the gap in throughput reduces from 43% to 25%. As in the previous experiments, Zyzzyva yields a significantly lower throughput as it cannot handle failures. In case of HotStuff, an increase in batch size does increases its throughput but due to high scaling of the graph this change seems insignificant.

5.6 Disabling Out-of-Ordering

Until now, we allowed protocols like Pbft, PoE, SBFT and Zyzzyva to process requests *out-of-order*. As a result, these protocols achieve much higher throughputs than HotStuff, which is restricted by its sequential primary-rotation model. In Figures 9(k) and 9(l), we evaluate the performance of the protocols when there are no opportunities for out-of-ordering.

In this setting, we require each client to only send its request when it has accepted a response for its previous query. As HotStuff pipelines its phases of consensus into a *four*-phase pipeline, so we allow it to access four client requests (each on a distinct subsequent replica) at any time. As expected, HotStuff performs better than all other protocols at the expense of a higher latency as it rotates primaries at the end of each consensus, which allows

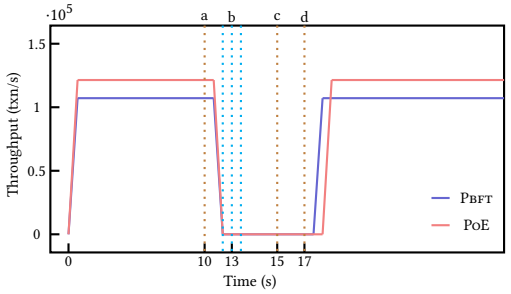


Figure 10: System throughput under instance failures (n = 32). (a) replicas detect failure of primary and broadcast VC-REQUEST; (b) replicas receives VC-REQUEST from others; (c) replicas receives NV-PROPOSE from new primary; (d) state recovery;

it to pipeline four requests. However, notice that once out-of-ordering is disabled, throughput drops from 200 ktransactions/s to just under a few *thousand* transactions/s. Hence, from a practical standpoint, out-of-ordering is simply crucial. Further, the difference in latency of different protocols is quite small, and the visible variation is a result of graph scaling while the actual numbers are in the range of 20 ms–40 ms.

5.7 Primary Failure–View Change

In Figure 10, we study the impact of of a benign primary failure on PoE and Pbft. To recover from a primary failure, backup replicas run the view-change protocol. We skip illustrating view-change plots for Zyzzyva and SBFT as they already face severe reduction in throughput for a single backup failure. Further, Zyzzyva has an *unsafe* view-change algorithm and SBFT's view-change algorithm is no less expensive than Pbft. For HotStuff, we do not

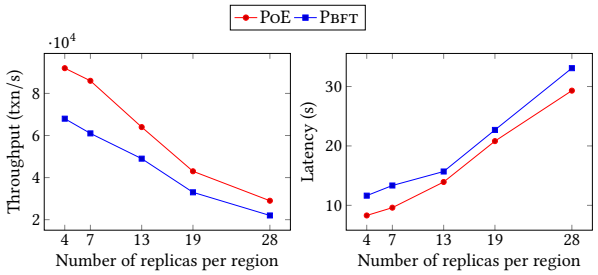


Figure 11: System throughput and average latency incurred by PoE and Pbft in a WAN deployment of five regions under a single failure. In the largest deployment, we have 140 replicas spread equally over these regions.

show results as it changes primary at the end of every consensus. Although single primary protocols face a momentary loss in throughput during view-change, these protocols easily cover this gap through their ability to process messages out-of-order.

For our experiments, we let the primary replica complete consensus for 10 s (or around a million transactions) and then fail. This causes clients to timeout while waiting for responses for their pending transactions. Hence, these clients forward their requests to backup replicas.

When a backup replica receives a client request, it forwards that request to the primary and waits on a timer. Once a replicas timeouts, it detects a primary failure and broadcasts a VC-REQUEST message to all other replicas—initiate view-change protocol (a). Next, each replica waits for a new view message from the next primary. In the meantime, a replica may receive VC-REQUEST messages from other replicas (b). Once a replica receives NV-PROPOSE message from the new primary (c), it moves to the next view.

5.8 WAN Scalability

In this section, we use Figure 11 to illustrate the throughputs and latencies for different PoE and Pbft deployments on a wide-area network in the presence of a single failure. In specific, we deploy clients and replicas across *five* locations across the globe: Oregon, Iowa, Montreal, the Netherlands, and Taiwan. Next, we vary the number of replicas from 20 to 140 by equally distributing these replicas across each region.

These plots affirm our existing observations that PoE outperforms existing state-of-the-art protocols and scales well in wide-area deployments. In specific, PoE achieves up to 1.41× higher throughput and incurs 28.67% less latency than Pbft. We skip presenting plots for SBFT, HotStuff and Zyzzyva due to their low throughputs under failures.

5.9 Simulating bft Protocols

To further underline that the *message delay* and not *bandwidth requirements* becomes a determining factor in the throughput of protocols in which the primary does not propose requests out-of-order, we performed a separate simulation of the maximum performance of PoE, Pbft, and HotStuff. The simulation makes 500 consensus decisions and processes all message send and receive steps, but delays the arrival of messages by a predetermined message delay. The simulation skips any expensive computations and, hence, the simulated performance is entirely determined by the cost of message exchanges. We ran the simulation with $\mathbf{n} \in \{4, 16, 128\}$ replicas, for which the results can be found in Figure 12, first three plots. As one can see, if bandwidth is not a limiting factor, then the performance of protocols that do not propose requests out-of-order will be determined by the number of communication rounds and the message delay. As both Pbft and PoE have one communication round more than the two rounds of HotStuff, their performance is roughly two-thirds that of HotStuff, this independent of the number of replicas or the message delay. Furthermore, doubling message delay will roughly half performance. Finally, we also measured the maximum performance of protocols that do allow out-of-order processing of up to 250 consensus decisions. These results can be found in Figure 12, last plot. As these results show, out-of-order processing increases performance by a factor of roughly 200, even with 128 replicas.

6 RELATED WORK

Consensus is an age-old problem that received much theoretical and practical attention (see, e.g., [34, 39, 45]). Further, the use of rollbacks is common in distributed systems. E.g., the crash-resilient replication protocol Raft [45] allows primaries to rewrite the log of any replica. In a Byzantine environment, such an approach would delegate too much power to the primary, as they can maliciously overwrite transactions that need to be preserved.

The interest in practical bft consensus protocols took off with the introduction of Pbft [9]. Apart from the protocols that we already discussed, there are some interesting protocols that achieve efficient consensus by requiring $5\mathbf{f} + 1$ replicas [1, 14]. However, these protocols have been shown to work only in the cases where transactions are non-conflicting [38]. Some other bft protocols [10, 50] suggest the use of *trusted components* to reduce the cost of bft consensus. These works require only $2\mathbf{f}+1$ replicas as the trusted component helps to guarantee a correct ordering. The safety of these protocols relies on the security of trusted component. In comparison, PoE does (i) not require extra replicas, (ii) not depend on clients, (iii) not require trusted components, and (iv) not need the two phases of quadratic communication required by Pbft.

As a promising future direction, Castro [9] also suggested exploring speculative optimizations for Pbft, which he referred to as tentative execution. However, this lacked: (i) formal description, (ii) non-divergence safety property, (iii) specification of rollback under attacks, (iv) re-examination of the view change protocol, and (v) any actual evaluation.

Consensus for Blockchains: Since the introduction of Bitcoin [42], the well-known cryptocurrency that led to the coining of the term blockchain, several new bft consensus protocols that cater to cryptocurrencies have been designed [33, 37]. Bitcoin [42] employs the *Proof-of-Work* [33] consensus protocol (PoW), which is computationally intensive, achieves low throughput, and can cause forks (divergence) in the blockchain: separate chains can exist on non-faulty replicas, which in turn can cause *double-spending attacks* [31]. Due to these limitations, several other similar algorithms have been proposed. E.g., *Proof-of-Stake* (PoS) [37], which is design such that any replica owning $n\%$ of the total resources gets the opportunity to create $n\%$ of the new blocks. As PoS is resource driven, it can face attacks where replicas are incentivized to work simultaneously on several forks of the blokchain, without ever trying to eliminate these forks.

There are also a set of interesting alternative designs such as ConFlux [40], Caper [3] and MeshCash [6] that suggest the use of directed acyclic graphs (DAGs) to store a blockchain to improve the performance of Bitcoin. However, these protocols either rely on PoW or Pbft for consensus.

Meta-protocols such as RCC [28] and RBFT [5] run multiple Pbft consensuses in parallel. These protocols also aim at removing dependence on the consensus led by a single primary. A recent protocol, PoV [41], provides fast bft consensus in a

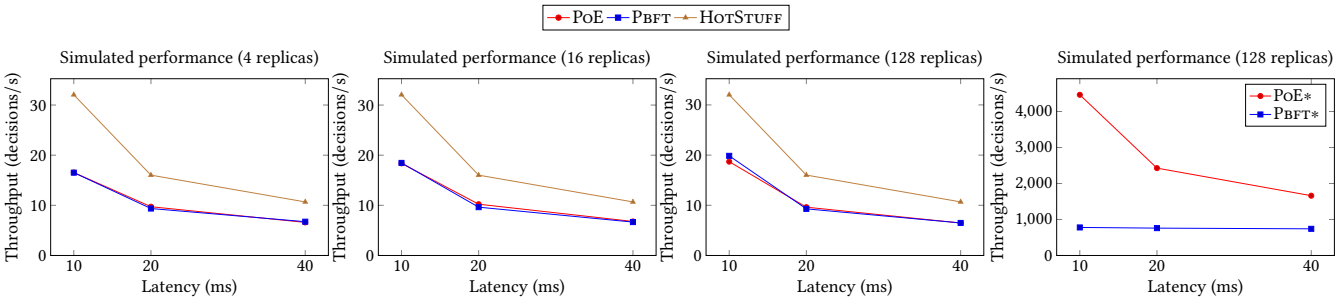


Figure 12: The simulated number of consensus decisions PoE, Pbft, and HotStuff can make as a function of the latency. Only the protocols in the right-most plot and marked with $*$ process requests out-of-order processing.

consortium architecture. PoV does this by restricting the ability to propose blocks among a subset of trusted replicas.

PoE does not face the limitations faced by PoW [33] and PoS [37]. The use of DAGs [3, 6, 40], and sharding [15, 52] is orthogonal to the design of PoE. Hence, their use with PoE can reap further benefits. Further, PoE can be employed by meta-protocols and does not restrict consensus to any subset of replicas.

7 CONCLUSIONS

We present Proof-of-Execution (PoE), a novel Byzantine fault-tolerant consensus protocol that guarantees safety and liveness and does so in only three linear phases. PoE decouples ordering from execution by allowing replicas to process messages out-of-order and execute client-transactions speculatively. Despite these properties, PoE ensures that all the replicas reach a single unique order for all the transactions. Further, PoE guarantees that if a client observes identical results of execution from a majority of the replicas, then it can reliably mark its transaction committed. Due to speculative execution, PoE may require replicas to revert executed transactions, however. To evaluate PoE's design, we implement it in our ResilientDB fabric. Our evaluation shows that PoE achieves up-to-80% higher throughputs than existing bft protocols in the presence of failures.

REFERENCES

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. 2005. Fault-scalable Byzantine Fault-tolerant Services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. ACM, 59–74. https://doi.org/10.1145/1095810.1095817

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. https://arxiv.org/abs/1712.01367

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. CAPER: A Cross-application Permissioned Blockchain. *Proc. VLDB Endow.* 12, 11 (2019), 1385–1398. https://doi.org/10.14778/3342263.3342275

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30:1–30:15. https://doi.org/10.1145/3190508.3190538

[5] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. 2013. RBFT: Redundant Byzantine Fault Tolerance. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 297–306. https://doi.org/10.1109/ICDCS.2013.53

[6] Iddo Bentov, Pavel Hubáček, Tal Moran, and Asaf Nadler. 2017. Tortoise and Hares Consensus: the Meshcash Framework for Incentive-Compatible, Scalable Cryptocurrencies. https://eprint.iacr.org/2017/300

[7] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 355–362. https://doi.org/10.1109/DSN.2014.43

[8] Manuel Bravo, Zsolt István, and Man-Kit Sit. 2020. Towards Improving the Performance of BFT Consensus For Future Permissioned Blockchains. https://arxiv.org/abs/2007.12637

[9] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461. https://doi.org/10.1145/571637.571640

[10] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested Append-Only Memory: Making Adversaries Stick to Their Word. *SIGOPS Oper. Syst. Rev.* 41, 6 (2007), 189–204. https://doi.org/10.1145/1323293.1294280

[11] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. Upright Cluster Services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 277–290. https://doi.org/10.1145/1629575.1629602

[12] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 153–168.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[14] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX, 177–190.

[15] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 123–140. https://doi.org/10.1145/3299869.3319889

[16] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1085–1100. https://doi.org/10.1145/3035918.3064033

[17] Wayne W. Eckerson. 2002. *Data quality and the bottom line: Achieving Business Success through a Commitment to High Quality Data*. Technical Report. The Data Warehousing Institute, 101communications LLC.

[18] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB: A Shared Database on Blockchains. *Proc. VLDB Endow.* 12, 11 (2019), 1597–1609. https://doi.org/10.14778/3342263.3342636

[19] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382. https://doi.org/10.1145/3149.214121

[20] Gideon Greenspan. 2015. MultiChain Private Blockchain–White Paper. https://www.multichain.com/download/MultiChain-White-Paper.pdf

[21] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 568–580. https://doi.org/10.1109/DSN.2019.00063

[22] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, 393–481. https://doi.org/10.1007/3-540-08755-9_9

[23] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2019. An In-Depth Look of BFT Consensus in Blockchain: Challenges and Opportunities. In *Proceedings of the 20th International Middleware Conference Tutorials, Middleware*. ACM, 6–10. https://doi.org/10.1145/3366625.3369437

[24] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2020. Blockchain consensus unraveled: virtues and limitations. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. ACM, 218–221. https://doi.org/10.1145/3401025.3404099

[25] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2020. Building High Throughput Permissioned Blockchain Fabrics: Challenges and Opportunities. *Proc. VLDB Endow.* 13, 12 (2020), 3441–3444. https://doi.org/10.14778/3415478.3415565

[26] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2019. Brief Announcement: Revisiting Consensus Protocols through Wait-Free Parallelization. In *33rd International Symposium on Distributed Computing (DISC 2019)*, Vol. 146. Schloss Dagstuhl, 44:1–44:3. https://doi.org/10.4230/LIPIcs.DISC.2019.44

[27] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. *Fault-Tolerant Distributed Transactions on Blockchain*. Morgan & Claypool. https://doi.org/10.2200/S01068ED1V01Y202012DTM065

[28] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *37th IEEE International Conference on Data Engineering*. IEEE. to appear.

[29] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (2020), 868–883. https://doi.org/10.14778/3380750.3380757

[30] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. 2020. Permissioned Blockchain Through the Looking Glass: Architectural and Implementation Lessons Learned. In *Proceedings of the 40th IEEE International Conference on Distributed Computing Systems*.

[31] Suyash Gupta and Mohammad Sadoghi. 2019. Blockchain Transaction Processing. In *Encyclopedia of Big Data Technologies*. Springer, 1–11. https://doi.org/10.1007/978-3-319-63962-8_333-1

[32] Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. 2007. *Data Quality and Record Linkage Techniques*. Springer. https://doi.org/10.1007/0-387-69505-2

[33] Markus Jakobsson and Ari Juels. 1999. Proofs of Work and Bread Pudding Protocols. In *Secure Information Networks: Communications and Multimedia Security IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS'99)*. Springer, 258–272. https://doi.org/10.1007/978-0-387-35568-9_18

[34] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*. IEEE, 245–256. https://doi.org/10.1109/DSN.2011.5958223

[35] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. USENIX, 237–250.

[36] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography* (2nd ed.). Chapman and Hall/CRC.

[37] Sunny King and Scott Nadal. 2012. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake. https://www.peercoin.net/whitepapers/peercoin-paper.pdf

[38] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2009. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.* 27, 4 (2009), 7:1–7:39. https://doi.org/10.1145/1658357.1658358

[39] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (2001), 51–58. https://doi.org/10.1145/568425.568433 Distributed Computing Column 5.

[40] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. 2018. Scaling Nakamoto Consensus to Thousands of Transactions per Second. https://arxiv.org/abs/1805.03870

[41] Kejiao Li, Hui Li, Han Wang, Huiyao An, Ping Lu, Peng Yi, and Fusheng Zhu. 2020. PoV: An Efficient Voting-Based Consensus Algorithm for Consortium Blockchains. *Front. Blockchain* 3 (2020), 11. https://doi.org/10.3389/fbloc.2020.00011

[42] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf

[43] Faisal Nawab and Mohammad Sadoghi. 2019. Blockplane: A Global-Scale Byzantizing Middleware. In *35th International Conference on Data Engineering (ICDE)*. IEEE, 124–135. https://doi.org/10.1109/ICDE.2019.00020

[44] The Council of Economic Advisers. 2018. *The Cost of Malicious Cyber Activity to the U.S. Economy*. Technical Report. Executive Office of the President of the United States. https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf

[45] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX, 305–320.

[46] M. Tamer Özsu and Patrick Valduriez. 2020. *Principles of Distributed Database Systems*. Springer. https://doi.org/10.1007/978-3-030-26253-2

[47] Sajjad Rahnama, Suyash Gupta, Thamir Qadah, Jelle Hellings, and Mohammad Sadoghi. 2020. Scalable, Resilient and Configurable Permissioned Blockchain Fabric. *Proc. VLDB Endow.* 13, 12 (2020), 2893–2896. https://doi.org/10.14778/3415478.3415502

[48] Thomas C. Redman. 1998. The Impact of Poor Data Quality on the Typical Enterprise. *Commun. ACM* 41, 2 (1998), 79–82. https://doi.org/10.1145/269012.269025

[49] Dale Skeen. 1982. *A Quorum-Based Commit Protocol*. Technical Report. Cornell University.

[50] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (2013), 16–30. https://doi.org/10.1109/TC.2011.221

[51] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 347–356. https://doi.org/10.1145/3293611.3331591

[52] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: Scaling Blockchain via Full Sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 931–948. https://doi.org/10.1145/3243734.3243853

Scalable Linear Algebra Programming for Big Data Analysis

Leonidas Fegaras
University of Texas at Arlington
Arlington, Texas
fegaras@cse.uta.edu

ABSTRACT

Arrays are very important data structures for many data-centric and scientific applications. One of the most effective representations of large dense arrays in a distributed setting is a block array, such as a tiled matrix, which is a distributed collection of non-overlapping dense array blocks. Although there are many linear algebra libraries for machine learning that support distributed block arrays and provide an optimal implementation for many array operations, these libraries do not support ad-hoc array programming and customized storage structures. Imperative programs with loops and array indexing, on the other hand, are more powerful as they allow arbitrary array computations but are hard to parallelize and convert to distributed programs.

Our goal is to provide an SQL-like abstraction for data-parallel distributed array computations that is expressive enough to capture a large class of array computations and can be compiled to efficient data-parallel distributed code. Our abstraction is a monolithic array construction in the form of an array comprehension that is as expressive as SQL by supporting a group-by syntax that allows us to capture many array computations in declarative form. We present rules for translating array comprehensions on block arrays to data-parallel distributed code that can run on Apache Spark. We describe a comprehensive set of effective optimizations that can produce very efficient translations, such as the optimal block matrix multiplication algorithm, even though they are oblivious to linear algebra operations. Finally, we justify our claims by evaluating the performance of our generated code on Apache Spark relative to Spark MLlib.

1 INTRODUCTION

Much of the data used in data-centric applications come in the form of arrays, such as vectors, matrices, and tensors. In the early days of numerical computing, most of the array programming was done in an imperative loop-based language, such as Fortran or C, using array indexing to access and update array elements incrementally, one at a time. Although loop-based programs are efficient when they run on a single processor, they are hard to parallelize and reason about. Currently, most array programming is done using vectorization languages, such as MATLAB, R, and NumPy, that allow programmers to write high-level array code that closely resembles mathematical formulas. These languages provide highly tuned array operations that are applied to whole arrays instead of individual elements, thus making loops inessential. Moreover, they hide the implementation details and optimize performance by choosing an implementation (a kernel) for an array operation from a variety of build-in array storages and algorithms. Internally, these languages rely on numerical libraries, such as BLAS [9], for efficient linear algebra computations. These libraries, which are also an integral part of many machine learning (ML) sytems, such as TensorFlow [1], PyTorch, and MLlib [5], implement basic array operations efficiently using multicore parallelism and GPU acceleration. Many array operations provided by the vectorization languages are overloaded to work on a variety of array storage structures, thus offering an implementation-independent view to the programmer. Given that there are numerous storage structures for arrays, such as dense, tiled, and compressed sparse matrices, each library operation must have numerous implementations, especially those operations that operate on multiple arrays, such as matrix multiplication. As a result, this code specialization based on array implementation is hard to extend with user-defined storage structures and algorithms. This is particularly true for distributed arrays, which have to be partitioned into blocks and distributed across compute nodes. In that case, not only there are numerous ways to implement these blocks as arrays, but there are also numerous ways to partition the arrays into blocks. A better solution would have been to express array computations in a high-level declarative language for array computations that is expressive enough to capture most array programs and is supported by a translation scheme that separates the specification from the implementation and generates high-quality code.

This problem of sacrificing expressiveness and extensibility for efficiency incurred from the library approach is exacerbated by the need to process large arrays that do not fit in memory. Given that the accuracy of the data analysis and ML models depends on the data size, current data-centric applications must analyze enormous amounts of array data using complex mathematical data processing methods. In recent years, new frameworks in distributed Big Data analytics have become essential tools for large-scale machine learning and scientific discoveries. These systems, which are also known as Data-Intensive Scalable Computing (DISC) systems, have revolutionized our ability to analyze Big Data. Unlike High-Performance Computing (HPC) systems, which are mainly designed for shared-memory architectures, DISC systems are distributed data-parallel systems on clusters of shared-nothing computers connected through a high-speed network. Compared to low-level distributed-memory communication paradigms, such as MPI, DISC systems automate many aspects of distributed computing, such as fault tolerance, which is important for long-running Big Data analysis on thousands of computers, scalability, data partitioning and distribution, and task scheduling and management. One of the earliest DISC systems is Map-Reduce [8], which was introduced by Google and later became popular as an open-source software with Apache Hadoop. Recent systems, such as Apache Spark [4] and Apache Flink [3], go beyond Map-Reduce by maintaining dataset partitions in the memory of the compute nodes. All these systems use data shuffling to exchange data among compute nodes, which takes place implicitly between the map and reduce stages in Map-Reduce and during group-bys and joins in Spark and Flink. Essentially, all data exchanges across compute nodes are done in a controlled way using special operations, which implement data shuffling by

distributing data based on some key, so that data associated with the same key are processed together by the same compute node.

The goal of this paper is to provide a well-formed abstraction for data-parallel distributed array computations "without regret", that is, an abstraction that is declarative so that we can reason about it, is expressive enough to capture a large class of array computations, and can be compiled to efficient data-parallel distributed code. Our main construct is the *array comprehension*, which is a monolithic array construction in the form of a list comprehension. List comprehensions are found in many modern programming languages, such as Python, Scala, and Haskell. Unlike regular list comprehensions though, our array comprehensions are as expressive as SQL queries by supporting a group-by syntax that allows us to capture many array computations in declarative form without using array indexing which is hard to reason about. An array comprehension can access and correlate multiple arrays by traversing their elements one-by-one and can construct a new array in one shot by mapping array indices to values, which are derived from the elements of the input arrays. Array comprehensions can capture many linear algebra operations, including inner and outer products of vectors, matrix addition and multiplication, matrix rotation and transpose, array slicing and concatenation. More complex array operations, such as matrix inverse and LU decomposition, can be coded using array comprehensions inside loops.

1.1 Highlights of our Approach

This paper presents a generic and customizable system that translates abstract array programs to high-performance distributed code that can run on current DISC platforms. When designing storage structures for arrays in a distributed setting, there are many choices to consider, each exhibiting different performance characteristics for various array computations. One example of such a storage method is organizing contiguous array elements into dense non-overlapping blocks of fixed capacity. Our framework uses a two-layer approach where array programs are expressed in a powerful high-level syntax on abstract arrays, while these abstract arrays are mapped to customized storage structures based on user-defined type mappings, thus separating specification from implementation.

An abstract array with dimensionality i in our framework has type array$i$[T], for an arbitrary type T. The most common abstract arrays are vector[T], equal to array1[T], and matrix[T], equal to array2[T]. An abstract array is represented as an association list of key-value pairs in which the key contains the array indices. This array representation is also known as a sparse representation or a coordinate format. For example, a matrix M of type matrix[Double] is represented as an association list of type List[((Int,Int),Double)] so that an element M_{ij} is represented by the key-value pair $((i,j),M_{ij})$, which associates the indices i and j with the value M_{ij}. This association list can be sparse (i.e., some elements may be missing) if the array is sparse. A concrete implementation of an array (i.e., its storage structure) is specified by two customized functions: the *sparsifier*, which converts the storage structure to an association list, and the *builder*, which constructs the storage structure from the association list. These two functions, which are inverse of each other, are used by our translator to transform any operation $f(x_1,\ldots,x_n)$ on abstract arrays x_i to an operation on their concrete storage structures c_i by up-coercing the storages c_i using the sparsifiers s_i and down-coercing the abstract result using the builder b, that is,

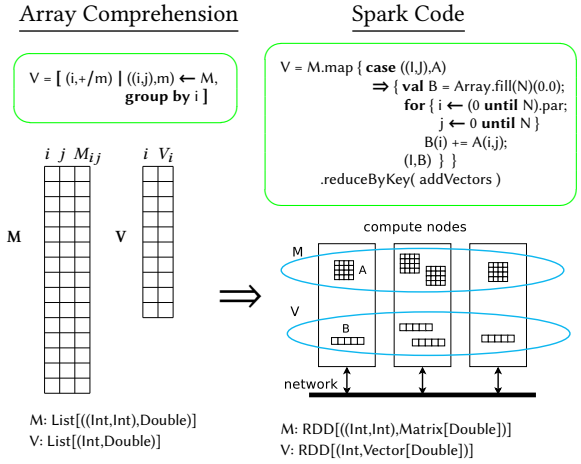


Figure 1: Code generation for $V_i = \sum_j M_{ij}$

$b(f(s_1(c_1),\ldots,s_n(c_n)))$. This layered approach introduces levels of indirection and generates superfluous intermediate structures (the abstract arrays x_i) that need to be removed. In our framework, this is accomplished by fusing these functions into one function that represents the concrete code so that the resulting program builds the output structures directly and works on the storage structures c_i without creating the association lists x_i.

Our language for expressing abstract array programs uses a monolithic array construction in the form of an array comprehension that is as expressive as SQL by supporting a group-by syntax that allows us to capture many array computations in declarative form. Comprehensions with a group-by syntax were first introduced by Wadler and Peyton Jones [24] and have been used as the formal calculus for the DISC query languages MRQL [10] and DIQL [12]. For example, the following comprehension returns the number of employees in each department:

```
[ (d.name,count(e)) | e ← Employees, d ← Departments,
                      e.dno == d.dnumber, group by d.name ].
```

The formal semantics of comprehensions, the translation of comprehensions to an algebra, and a query optimization framework are described in our earlier work [10, 12]. Array comprehensions are actually list comprehensions in which the array storages used in a comprehension are implicitly converted to association lists by array sparsifiers and the list returned by the comprehension is converted to an array storage by an array builder.

Consider for example the following array comprehension that constructs a vector V of size n from a matrix M of size $n \times m$ such that $V_i = \sum_j M_{ij}$, where both the matrix M and the resulting vector V are stored in memory:

```
V = vector(n)[ ( i, +/m ) | ((i,j),m) ← M, group by i ],        (1)
```

which constructs the entire array V in one shot. The matrix M of type matrix[Double] is implicitly converted to an association list of type List[((Int,Int),Double)]. The generator ((i,j),m) ← M traverses M one element at a time and each time the traversed element is pattern-matched with the pattern ((i,j),m), which binds the pattern variables i, j, and m to the the corresponding components of this element. A group-by operation in a comprehension lifts each pattern variable defined before the group-by (except the group-by keys) from some type t to a bag of t, indicating that each such variable must now contain all the values associated with the same group-by key value. Consequently, after we group

by i, the pattern variables that are not used in the group-by key, j and m, are now lifted to lists that contain all their values associated with a certain group-by key i. The term +/m adds up all the bindings of the variable m associated with the key i. That is, it calculates $\sum_j M_{ij}$. Finally, the array builder, vector(n)(L), converts the association list L of type List[(Int,Double)] to a vector of type vector[Double] of size n.

In Query (1), both matrix M and the query result V are stored in memory. However, in our framework, the storage of these arrays can be customized by using a different sparsifier for M and a different builder for the result. More specifically, we want to translate array comprehensions to efficient distributed programs over block arrays, which are distributed bags of dense array chunks. In Spark [4], we can implement a tiled matrix as a distributed collection (an RDD) of fix-sized square tiles of type RDD[((Int,Int),Array[Double])], where each tile ((i,j),A) has coordinates i and j and values stored in the dense matrix A, which has a fixed size N*N, for some constant N. Similarly, we can implement the query result V as a block vector of type RDD[(Int,Array[Double])], where each block (i,B) has a coordinate i and values stored in the vector B of size N. Then, Query (1) can be rewritten as:

V = tiled[(i, +/m) | ((i,j),m) ← M, **group by** i]. (2)

For this query, the implicit sparsifier that converts a tiled matrix M of type RDD[((Int,Int),Array[Double])] to an association list of type List[((Int,Int),Double)] is:

```
[ ( ( ii*N+i, jj*N+j ), A(i,j) )
| ((ii,jj),A) ← M, i ← 0 until N, j ← 0 until N ],
```

where the index variable i in 'i ← 0 **until** N' iterates from 0 to N−1. Here, ii and jj are tile coordinates, and i and j are indices within a tile. That is, for each tile A with coordinates ii and jj read from the tiled matrix M, this list comprehension constructs $N * N$ elements from the data stored in A, so that each element has row coordinate ii*N+i, column coordinate jj*N+j, and value A(i,j). On the other hand, the builder tiled(L), which constructs a block array of type RDD[(Int,Array[Double])] from the list L of type List[(Int,Double)], is:

```
rdd[ ( i/N, vector(N)(w) )
   | (i,v) ← L, let w = ( i%N, v ), group by i/N ],
```

where the builder rdd builds an RDD from a list. That is, this comprehension groups the elements (i,v) of L by i/N (the tile coordinate) so that all N pairs (i,v) with the same i/N go into the same tile. After the group-by, the value of w is lifted to a list that contains all the values (i%N,v) that belong to the same tile at location i/N. The builder vector(N)(w), used in Query (1), converts the list w to a vector, which is an array block of size N.

Like array queries, sparsifiers and builders are expressed as comprehensions, but unlike array queries, they use efficient array indexing. This implicit coercion from stored arrays to lists and the building of stored arrays from the results of a comprehension introduce levels of indirection and generate superfluous intermediate structures that need to be removed. This is done by unnesting nested comprehensions into flat comprehensions. As we will show in this paper, after some simple transformations, Query (2) is optimized to the Spark code shown on the right of Figure 1. That is, from every tile A in M with tile coordinates I and J, a new vector block B with a coordinate I is constructed. The tile processing, which takes place at each compute node, is parallelized using the the Scala's par method [20]. Finally, vector blocks with the same coordinate are reduced pairwise using vector addition, addVectors.

In this paper, we present a small set of generic rules for translating array comprehensions to efficient Spark RDD programs that operate on block arrays. These rules are not based on specific array operations but can apply to many array processing programs that can be expressed as array comprehensions. Matrix multiplication, for example, is translated to an optimal block matrix multiplication algorithm by one generic rule that recognizes a certain class of group-by-joins (joins between two datasets followed by a group-by and an aggregation), and translates them to a special block group-by-join algorithm. Our system, called *SAC* (Scalable Array Comprehensions), has been implemented using Scala's compile-time reflection and macros. It translates array comprehensions to Scala code that calls Spark RDD operations whose functional arguments use the Scala's Parallel Collections library for multicore parallelism [20].

In an earlier work [13], we have presented a framework, called DIABLO (a Data-Intensive Array-Based Loop Optimizer), for translating array-based loops to array comprehensions, which in turn are translated to Spark programs expressed in the Spark Core API. The DIABLO input language resembles the syntax of some loop-based imperative languages, such as C and Java. DIABLO can translate any array-based loop expressed in this loop-based language to an equivalent Spark program as long as this loop satisfies some simple syntactic restrictions, which are more permissive than the recurrence restrictions imposed by many current systems. Unlike SAC, DIABLO generates Spark programs that operate on arrays in the coordinate format. In a distributed setting, arrays stored in the coordinate format are known to be less efficient than block arrays because 1) they occupy more space and therefore require more data shuffling to evaluate complex array operations, and 2) they are less amenable to multicore parallelism at each compute node since they store the array elements in random order. Furthermore, the focus of DIABLO is in the translation of imperative programs to array comprehensions, while the focus of SAC is in the translation of array comprehensions to efficient code on customizable array storages, with an emphasis on block arrays. That is, SAC supplements DIABLO and can be used as a drop-in back-end replacement for DIABLO to make it able to work on block arrays. Finally, the work reported in this paper generalizes our earlier work on extending MRQL with array-based computations [11]. It extends the MRQL query optimizer with a GroupByJoin physical operator that generalizes the SUMMA parallel algorithm for matrix multiplication [14], an idea also used in Section 5.4 in the context of block arrays. However, unlike our current framework, this system too is based on arrays stored in the coordinate format.

The contributions of this paper are summarized as follows:

- We introduce a novel comprehension syntax for array computations that can capture many array computations in declarative form and is independent of array storage.
- We describe a translation scheme that translates array comprehensions to efficient imperative programs with memory effects (Sections 2 and 3).
- We extend this translation scheme to generate efficient Spark code from array comprehensions (Section 4).
- We introduce special type transformations to translate comprehensions on block arrays to efficient Spark code that reduces the amount of data shuffling (Section 5).

Expression:

e	::=	$[\, e \mid \overline{q}\,]$	comprehension
	|	$\oplus/e$	reduction using $\oplus$
	|	$v[e_1, \ldots, e_n]$	array indexing $n \geq 1$
	|	$\ldots$	other expression

Qualifiers:

$\overline{q}$	::=	$q_1, \ldots, q_n$	$n \geq 0$

Qualifier:

q	::=	$p \leftarrow e$	generator
	|	**let** $p = e$	local declaration
	|	e	filtering
	|	**group by** $p\,[: e\,]$	group-by

Pattern:

p	::=	v	pattern variable
	|	$(p_1, \ldots, p_n)$	tuple $n \geq 0$

Figure 2: Language syntax

$$[\, e_1 \mid p \leftarrow e_2, \overline{q}\,] = e_2.\text{flatMap}(\lambda p.\, [\, e_1 \mid \overline{q}\,]) \quad (4)$$

$$[\, e_1 \mid \textbf{let}\, p = e_2, \overline{q}\,] = \textbf{let}\, p = e_2\, \textbf{in}\, [\, e_1 \mid \overline{q}\,] \quad (5)$$

$$[\, e_1 \mid e_2, \overline{q}\,] = \textbf{if}\, (e_2)\, \textbf{then}\, [\, e_1 \mid \overline{q}\,]\, \textbf{else}\, \text{Nil} \quad (6)$$

$$[\, e \mid\,] = [\, e\,] \quad (7)$$

Figure 3: Desugaring rules

- We evaluate the performance of our system relative to Spark's MLlib.linalg library (Section 6). Based on these results, SAC is up to 6 times faster than MLlib for matrix multiplication and up to 3 times faster than MLlib for matrix factorization.

2 SYNTAX AND SEMANTICS OF ARRAY COMPREHENSIONS

Figure 2 describes the syntax of our language and Figure 3 gives the desugaring rules, which are based on standard methods for translating list comprehensions [23]. The meaning, desugaring rules, and code generation for the group-by syntax are given in Section 3.

Flattening nested comprehensions that do not have a group-by qualifier is done using the following rule:

$$[\, e_1 \mid \overline{q_1},\, p \leftarrow [\, e_2 \mid \overline{q_3}\,],\, \overline{q_2}\,] = [\, e_1 \mid \overline{q_1},\, \overline{q_3},\, \textbf{let}\, p = e_2,\, \overline{q_2}\,] \quad (3)$$

for any sequence of qualifiers $\overline{q_1}$, $\overline{q_2}$, and $\overline{q_3}$. It may require renaming the variables in $[\, e_2 \mid \overline{q_3}\,]$ before we apply this rule to prevent variable capture.

As explained in Section 1.1, abstract arrays in our framework are represented as association lists that uniquely map array indices to values. These abstract representations are mapped to concrete storage structures with the help of a pair of customized functions, a sparsifier and a builder. Then, array comprehensions on abstract arrays are translated to efficient concrete programs on storage structures based on these type mappings. In this and the following section, we describe this program translation in more detail using one specific type mapping that stores a matrix in a flat vector in row-major order. Although this storage structure is not a distributed tiled array, which is the focus of this paper, this example is important for two reasons: First, it illustrates our program translation process in detail using a simpler storage. Second, it is useful for translating comprehensions on block arrays to Spark code (described in Sections 4 and 5) because it shows how the code for tile operations is generated, given that a tile is a matrix.

Consider a matrix M of type Matrix[T] stored in row-major order as a triple (n,m,V) of type (Int,Int,Array[T]), where n and m are the matrix dimensions and V is the vector that contains the matrix elements in row-major order. The following sparsifier converts the storage S of type (Int,Int,Array[T]) to the abstract representation of the matrix M, which is of type List[((Int,Int),T)]:

```
def sparsify[T] ( S: (Int,Int,Array[T]) ): List[((Int,Int),T)]
= [ ((i,j),A(i*n+j)) | let (n,m,A) = S, i ← 0 until n, j ← 0 until m ],
```

where A(i) is array indexing in Scala. The builder matrix(n,m) L takes two groups of parameters. The n and m parameters specify the matrix dimensions, while L is the association list to be converted to a flat vector that contains the matrix values in row-major order:

```
def matrix[T] ( n: Int, m: Int )
              ( L: List[((Int,Int),T)] ): (Int,Int,Array[T])
= { val V = Array.ofDim[T](n*m);
    [ V(i*n+j) = v | ((i,j),v) ← L, i≥0, i<n, j≥0, j<m ];
    (n,m,V) },
```

where Array.ofDim[T](n) creates a new array of size n and V(i) = a is an assignment that updates V(i). The sparsifier function is always named 'sparsify' because, as we will see next, it is implicitly embedded in the code by the compiler by looking at the code type, while the builder must have a unique name or type signature since all builders transform association lists. Nevertheless, the builder too can be inferred by the compiler in certain assignments, such as in the following example. In the following declaration:

```
var M: matrix[Double]
  = matrix(n,m)[ ((i,j),random()) | i ← 0 until n, j ← 0 until m ];
```

the builder matrix(n,m) is required because the M declaration specifies the abstract type, matrix[Double], but not the storage. However, in the following assignment:

```
M = [ ((j,i),m+1) | ((i,j),m) ← M, m > 10 ];
```

the builder can be inferred to be matrix(n,m), since the compiler can infer the storage type of M.

As a running example to illustrate code generation, consider the addition of two matrices M and N of size $n \times m$, expressed as follows using array comprehensions:

```
matrix(n,m)[ ((i,j),a+b) | ((i,j),a) ← M, ((ii,jj),b) ← N,
                           ii == i, jj == j ].                    (8)
```

This query can also be expressed as:

```
matrix(n,m)[ ((i,j),a+N[i,j]) | ((i,j),a) ← M ],
```

which is translated to Query (8). Basically, an array indexing $V[e_1, \ldots, e_n]$ in a comprehension is transformed by adding the qualifiers $((k_1, \ldots, k_n), k_0) \leftarrow V, k_1 == e_1, \ldots, k_n == e_n$ to the comprehension, where $k_0, k_1, \ldots, k_n$ are fresh variables, and by replacing $V[e_1, \ldots, e_n]$ with k_0. Without such translation, array indexing would not be able to map to operations on the underlying array storage.

Given a generator $p \leftarrow e$ in a comprehension, the compiler will infer the type of e using standard type inference. Then, it will

search all defined sparsifiers to find one, if exists, that applies to the type of e, and will embed this sparsifier by replacing e with sparsify(e). For Query (8), the compiler will infer the storage type of M and N to be (Int,Int,Array[Float]) and then will embed the right sparsifiers to convert them to association lists, which for these matrices is the sparsify function defined earlier. Then, it will inline the code of the sparsifiers and builder and will optimize the resulting program. This can be done effectively when these functions are expressed as comprehensions. By expressing these functions as comprehensions, the optimizer can fuse them with the array comprehension of the query, resulting to a comprehension that traverses the array storage directly, without creating the intermediate lists. Furthermore, unlike array comprehensions, these functions can and must use array indexing so that the fused comprehension results to efficient array operations.

In addition to flattening nested comprehensions using Rule (3), the only optimizations needed are those related to index traversals. More specifically, if two index generators i ← 0 **until** n and j ← 0 **until** m are related with i==j, then they are fused to one generator and a let-binding: i ← 0 **until** min(n,m), **let** j = i.

For Query (8), if for simplicity, the inequalities in the matrix(n,m) builder are ignored, we have:

```
matrix(n,m)[ ((i,j),a+b) | ((i,j),a) ← sparsify(M),
                           ((ii,jj),b) ← sparsify(N),
                           ii == i, jj == j ]
    (if we inline the array builder without the inequalities)
= { val V = Array.ofDim[T](n*m);
    [ V(i*n+j) = v
    | ((i,j),v) ← [ ((i,j),a+b) | ((i,j),a) ← sparsify(M),
                                  ((ii,jj),b) ← sparsify(N),
                                  ii == i, jj == j ];
    (n,m,V) }
    (if we unnest the comprehension using Rule (3))
= { val V = Array.ofDim[T](n*m);
    [ V(i*n+j) = v
    | ((i,j),a+b) | ((i,j),a) ← sparsify(M),
                  ((ii,jj),b) ← sparsify(N),
                  ii == i, jj == j ];
    (n,m,V) }
    (if we inline the sparsifiers and rename their variables)
= { val V = Array.ofDim[T](n*m);
    [ V(i*n+j) = v
    | ((i,j),a+b) | ((i,j),a) ← [ ((i1,j1),A(i1*n1+j1))
                                | let (n1,m1,A) = M,
                                  i1 ← 0 until n1,
                                  j1 ← 0 until m1 ],
                  ((ii,jj),b) ← [ ((i2,j2),B(i2*n2+j2))
                                | let (n2,m2,B) = N,
                                  i2 ← 0 until n2,
                                  j2 ← 0 until m2 ],
                  ii == i, jj == j ];
    (n,m,V) }
    (if we unnest the comprehension using Rule (3))
= { val V = Array.ofDim[T](n*m);
    [ V(i*n+j) = v
    | let (n1,m1,A) = M,
      i1 ← 0 until n1, j1 ← 0 until m1,
      let (n2,m2,B) = N,
      i2 ← 0 until n2, j2 ← 0 until m2,
      i2 == i1, j2 == j1,
      let ((i,j),v) = ((i1,j1),A(i1*n1+j1)+B(i2*n2+j2)) ];
    (n,m,V) }
    (if we merge the array index bounds)
= { val V = Array.ofDim[T](n*m);
    [ V(i*n+j) = A(i1*n1+j1)+B(i1*n2+j1))
    | let (n1,m1,A) = M, let (n2,m2,B) = N,
      i1 ← 0 until min(n1,n2), j1 ← 0 until min(m1,m2) ];
    (n,m,V) }
```

Given that comprehensions over arrays are translated to array index traversals of type scala.collection.immutable.Range in Scala, to parallelize the code, the only transformation needed is to convert the outer index traversal to a parallel traversal of type scala.collection.parallel.immutable.ParRange. This is done by applying par, such as i1 ← (0 **until** min(n1,n2)).par in our example.

Comprehensions can also be used along with total aggregations, such as for checking whether a vector V is sorted:

```
&&/[ v <= w | (i,v) ← V, (j,w) ← V, j == i+1 ],
```

which checks if all consecutive elements of V (i.e., V_i and V_{i+1}) are ordered. The builder of $\oplus/e$ is:

```
{ var b = 1⊕; [ b = (b ⊕ v) | v ← e ]; b },
```

where $\mathbf{1}_\oplus$ is the zero value of the monoid $\oplus$. Similar to the matrix sparsifier, the vector sparsifier is:

```
def sparsify[T] ( V: Array[T] ): List[(Int,T)]
  = [ (i,V(i)) | i ← 0 until V.length ].
```

If we embed the sparsifiers, unfold the builder code, and unnest the list comprehensions, the array comprehension becomes:

```
{ var b = true;
  [ b = (b && (V(i) <= V(j))) | i ← 0 until V.length,
                                j ← 0 until V.length, j == i+1 ];
  b },
```

which is further optimized to:

```
{ var b = true;
  [ b = (b && (V(i) <= V(i+1))) | i ← 0 until V.length−1 ];
  b },
```

given that min(V.length,V.length−1) = V.length−1.

3 COMPREHENSIONS WITH A GROUP-BY

Consider the product of two matrices M and N with dimensions $n \times l$ and $l \times m$, respectively, which is equal to a matrix C so that $C_{ij} = \sum_k M_{ik} * N_{kj}$. Using our array comprehensions enhanced with a group-by syntax, matrix multiplication can be expressed as follows:

```
matrix(n,m)[ ((i,j),+/v) | ((i,k),a) ← M, ((kk,j),b) ← N,
                           kk == k, let v = a*b,
                           group by (i,j) ].                    (9)
```

This comprehension retrieves the values M_{ik} and N_{kj} and sets $v = M_{ik} * N_{kj}$. After we group the values by the matrix indices i and j, the variable v is lifted to a bag of numerical values $M_{ik} * N_{kj}$, for all k. Hence, the aggregation $+/v$ sums up all the values in the bag v, deriving $\sum_k M_{ik} * N_{kj}$ for the ij element of the resulting matrix.

Another example of a group-by comprehension is matrix smoothing for a matrix M, which is a matrix C such that $C_{ij} =$

$\frac{1}{9}\sum_{i-1\le I\le i+1}\sum_{j-1\le J\le j+1} M_{IJ}$. That is, C_{ij} is the average value in the neighborhood of M_{ij}:

```
matrix(n,m)[ ((ii,jj),(+/a)/a.length)
           | ((i,j),a) ← M,
             ii ← (i−1) to (i+1), jj ← (j−1) to (j+1),
             ii >= 0, ii < n, jj >= 0, jj < m,
             group by (ii,jj) ],
```

which also takes care of the boundary cases.

Given a pattern p that consists of bound pattern variables, the qualifier **group by** p in $[\, e \mid \overline{q_1},$ **group by** $p,\ \overline{q_2}\,]$ groups every pattern variable in $\overline{q_1}$ (except the variables in p) by the group-by key p into a list that contains all the values of this variable associated with this group-by key. Furthermore, the qualifier **group by** $p : e$ is syntactic sugar for the qualifiers **let** $p = e,$ **group by** p. Group-by qualifiers may appear in multiple places in a comprehension. For all these cases, only the pattern variables that precede the group-by qualifier in the same comprehension must be lifted to lists, and if multiple group-by qualifiers exist, these variables will have to be lifted multiple times to nested lists.

Group-by qualifiers are translated to groupBy operations before a comprehension is translated by the rules in Figure 2. Given a list s of type List$[(K,V)]$, the operation groupBy(s) groups the elements of s by their first component (the group-by key) and returns an association list of type Map$[K,$ List$[V]]$, which is implemented as a hash table:

```
def groupBy[K,V] ( s: List[(K,V)] ): Map[K,List[V]]
 = { val m = Map[K,List[V]]();
     [ m(k) = if (m.contains(k)) m(k):+v else List(v)          (10)
     | (k,v) ← s ];
     m }.
```

Let $\overline{v} = (v_1,\ldots,v_n)$ be the pattern variables in the sequence of qualifiers $\overline{q_1}$ that are used in the rest of the comprehension $[\, e \mid \overline{q_2}\,]$ but do not appear in the group-by pattern p. Then, the group-by syntax is translated as follows:

$$[\, e \mid \overline{q_1},\ \textbf{group by}\ p,\ \overline{q_2}\,] = [\, e \mid (p,s) \leftarrow \text{groupBy}([\, (p,\overline{v}) \mid \overline{q_1}\,]),\ \textbf{let}\ \overline{v} = \text{unzip}(s),\ \overline{q_2}\,], \quad (11)$$

where unzip$(s) = ([\, v_1 \mid \overline{v} \leftarrow s\,],\ldots,[\, v_n \mid \overline{v} \leftarrow s\,])$. That is, each pattern variable v_i in $\overline{v}$ is lifted to a list that contains all the values of v_i in the current group.

For example, the matrix multiplication in Query (9) has the following meaning:

```
matrix(n,m)[ ((i,j),(+/v)) | ((i,j),s) ← groupBy(S), let v = s ],
```

where S is:

```
[ ((i,j),v) | ((i,k),a) ← M, ((kk,j),b) ← N, kk == k, let v = a*b ]
```

since the only variable lifted is v with v = [v | v ← s], which is equal to s.

Array comprehensions with a group-by syntax allow more array operations to be expressed declaratively without having to use array indexing and loops. They also make comprehensions equivalent to basic SQL queries. As we will show, although it has pure semantics, the group-by syntax makes it easier to recognize certain code patterns in a comprehension and translate them to efficient code. For example, we will see next that the matrix multiplication in Query (9) is translated to the following code, which is as efficient as a program hand-coded in an imperative language:

```
{ val V = Array.ofDim[T](n*m)(0.0);
  [ V(i*n+j) += A(i*n+k)*B(k*l+j)
  | let (n,l,A) = M, let (ll,m,B) = N,
    i ← 0 until n, k ← 0 until l, j ← 0 until m ];
  (n,m,V) },
```

which is equivalent to a triple loop with body $V_{ij} \mathrel{+}= A_{ik} \times B_{kj}$.

Consider the general comprehension $[\, e \mid \overline{q_1},$ **group by** $p,\ \overline{q_2}\,]$. To simplify our translation rules, we rewrite this term to $[\, z \mid \overline{q_1},$ **group by** $p,\ z \leftarrow [\, e \mid \overline{q_2}\,]\,]$. Let $\mathcal{V} = \{v_1,\ldots,v_n\}$ be the set of variables that are lifted by the group-by but are not lifted or redefined in $\overline{q_2}$ and let $\overline{v} = (v_1,\ldots,v_n)$. Recall that these variables are lifted to lists that contain all the values of these variables associated with the group-by key p. A lifted variable may occur any number of times in the rest of the comprehension, $[\, e \mid \overline{q_2}\,]$. Let $w_1,\ldots,w_m$ be the occurrences of the lifted variables in $[\, e \mid \overline{q_2}\,]$. A lifted variable $w_i \in \mathcal{V}$ may occur in $[\, e \mid \overline{q_2}\,]$ as a term that takes one of the following forms:

- $\oplus_i/w_i$, for some monoid $\oplus_i$, or
- $\oplus_i/w_i.\text{map}(g_i)$, for some monoid $\oplus_i$ and a function g_i, or otherwise
- w_i, which is equal to $+\!\!+/w_i.\text{map}(x \Rightarrow \mathit{List}(x))$,

where the last case is used when the first two do not match. All these cases can be generalized to $\oplus_i/w_i.\text{map}(g_i)$, for some monoid $\oplus_i$ and function g_i. Hence, we can represent the term after group-by as follows:

$$\begin{aligned}[\, e \mid \overline{q_2}\,] &= f(\oplus_1/w_1.\text{map}(g_1),\ldots,\oplus_n/w_m.\text{map}(g_m)) \\ &= f(\otimes/\text{zip}(\overline{w}).\text{map}(g)),\end{aligned}$$

for some term function f, where $\overline{w} = (w_1,\ldots,w_m)$, $g = g_1 \times \cdots \times g_m$, and $\otimes = \oplus_1 \times \cdots \times \oplus_m$ (a product of monoids[1]). Then, based on the Rule (11) and the implementation of groupBy in definition (10), we have:

```
[ e | q̄1, group by p, q̄2 ]
= [ z | q̄1, group by p, z ← f(⊗/zip(w̄).map(g)) ]
= { val M = Map();                                            (12)
    [ M(p) = if (M.contains(p)) M(p) ⊗ g(w̄) else g(w̄) | q̄1 ];
    [ z | p ← M.keys, z ← f(M(p)) ] }
```

Consider now an array comprehension of the form:

$$\text{matrix}(n,m)[\, ((i,j),e) \mid \overline{q_1},\ \textbf{group by}\ (i,j),\ \overline{q_2}\,],$$

Notice that, here, the matrix index (i,j) in the comprehension head is the group-by key. In the implementation (12) of an array comprehension with group-by, we can now use arrays of size n*m, one array for each aggregation, instead of a Map:

```
matrix(n,m)[ ((i,j),e) | q̄1, group by (i,j), q̄2 ]
= matrix(n,m)[ z | q̄1, group by (i,j),
                 z ← f(⊕1/w1.map(g1),...,⊕n/wn.map(gn)) ]
= { val V1 = Array.fill(n*m)(1⊕1);
    ...
    val Vn = Array.fill(n*m)(1⊕n);
    [ { V1(i*n+j) = V1(i*n+j) ⊕1 g1(v1);
        ...
        Vn(i*n+j) = Vn(i*n+j) ⊕n gn(vn) } | q̄1 ];
    matrix(n,m)[ z | ((i,j),_) ← V1,
```

[1] That is, the monoid $\otimes$ with identity $\mathbf{1}_\otimes = (\mathbf{1}_{\oplus_1},\ldots,\mathbf{1}_{\oplus_m})$ and $(x_1,\ldots,x_m) \otimes (y_1,\ldots,y_m) = (x_1 \oplus_1 y_1,\ldots,x_m \oplus_m y_m)$.

$$z \leftarrow f(V_1(i*n+j),\ldots,V_n(i*n+j))\ ]\ \}.$$

Notice that, the final result is a matrix constructed using the matrix(n,m) builder, and is made out of the arrays that hold the aggregation results. This matrix though does not have a group-by and can be translated using the methods given in Section 2.

For example, the matrix multiplication in Query (9) is translated as follows:

```
matrix(n,m)[ ((i,j),+/v)
             | ((i,k),a) ← sparsify(M), ((kk,j),a) ← sparsify(N),
               kk == k, let v = a*b, group by (i,j) ],
    (after unfolding the sparsifiers for M and N)
= matrix(n,m)[ ((i,j),+/v)
             | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
               let (ll,m,B) = N, kk ← 0 until ll, j ← 0 until m,
               kk == k, let v = A(i*n+k)*B(kk*ll+j), group by (i,j) ]
    (after merging the array index kk with k)
= matrix(n,m)[ ((i,j),+/v)
             | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
               let (ll,m,B) = N, j ← 0 until m,
               let v = A(i*n+k)*B(k*ll+j), group by (i,j) ]
    (by translating the group-by qualifier)
= { val V = Array.fill(n,m)(0.0);
    [ V(i*n+j) = V(i*n+j) + A(i*n+k)*B(k*ll+j)
    | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
      let (ll,m,B) = N, j ← 0 until m ];
    matrix(n,m)[ v | ((i,j),_) ← V, v ← [ ((i,j),V(i*n+j)) |] ] }
= { val V = Array.fill(n,m)(0.0);
    [ V(i*n+j) = V(i*n+j) + A(i*n+k)*B(k*ll+j)
    | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
      let (ll,m,B) = N, j ← 0 until m ];
    matrix(n,m)[ ((i,j),V(i*n+j)) | ((i,j),_) ← V ] }
    (since the last term is equal to (n,m,V))
= { val V = Array.fill(n,m)(0.0);
    [ V(i*n+j) = V(i*n+j) + A(i*n+k)*B(k*ll+j)
    | let (n,l,A) = M, i ← 0 until n, k ← 0 until l,
      let (ll,m,B) = N, j ← 0 until m ];
    (n,m,V) }
```

which is equivalent to the desired efficient loop-based program.

4 TRANSLATING QUERIES ON SPARK

In this section, we translate array comprehensions to distributed programs that can run on Apache Spark [27]. Distributed datasets in Spark are represented as Resilient Distributed Datasets (RDDs), which support a functional API that is very similar to that for Scala collections. Most RDD operations are second-order, in which the functional argument is evaluated sequentially while the operation itself is evaluated in parallel, in a distributed mode. Unlike Scala collections, Spark does not allow nested RDDs and will raise a run-time error if the functional parameter of an RDD operation accesses another RDD. That is, Spark does not support nested parallelism because it is hard to implement efficiently in a distributed setting. However, instead of using nested RDD operations, one may use joins, cogroups, and cross products to correlate RDDs. Consequently, RDD comprehensions require special translation rules to derive joins, instead of nested flatMaps.

The RDD builder, rdd, that converts a List[T] to an RDD[T] can be implemented by applying the Spark method 'parallelize' on this list. However, RDD comprehensions must be translated to RDD operations in a special way to avoid generating nested operations. A group-by qualifier can be translated to the Spark groupByKey operation of type RDD[(K,V)]⇒RDD[(K,List[V])] using Equation (11). However, groupByKey is an expensive operation because it collects the grouped values into a list, shuffles these lists to the reducers, and finally reduces them by some aggregation. Instead, we want to generate calls to the more efficient reduceByKey(⊗), for some monoid ⊕, that reduces the values of type V using the monoid ⊗, instead of placing them into a list. That way, grouped values are partially reduced before they are shuffled. To generate these reduceByKey calls, we consider the group-by qualifier in combination with the aggregations in the comprehension. Recall that, based on the discussion in Section 3 and on Equation (12), any comprehension with a group-by can be put into the following form:

$$\begin{aligned}&\text{rdd}[\ e \mid \overline{q_1},\ \textbf{group by}\ p,\ \overline{q_2}\ ]\\ &\quad=\text{rdd}[\ z \mid \overline{q_1},\ \textbf{group by}\ p,\ z \leftarrow [\ e \mid \overline{q_2}\ ]\ ]\\ &\quad=\text{rdd}[\ z \mid \overline{q_1},\ \textbf{group by}\ p,\\ &\qquad z \leftarrow f(\oplus_1/w_1.\text{map}(g_1),\ldots,\oplus_m/w_m.\text{map}(g_m))\ ],\end{aligned}$$

for some variables w_i lifted by group-by, some monoids $\oplus_i$, and some functions g_i and f. Then, the group-by comprehension can be translated to a reduceByKey operation:

$$\begin{aligned}&\text{rdd}[\ e \mid \overline{q_1},\ \textbf{group by}\ p,\ \overline{q_2}\ ]\\ &\quad=\text{rdd}[\ (p,(g_1(w_1),\ldots,g_m(w_m))) \mid \overline{q_1}\ ]\\ &\qquad.\text{reduceByKey}(\otimes)\\ &\qquad.\text{map}\{\ \textbf{case}\ (p,(a_1,\ldots,a_m))\ \Rightarrow\ f(a_1,\ldots,a_m)\ \},\end{aligned}\tag{13}$$

where $\otimes = \oplus_1 \times \cdots \times \oplus_m$.

The following rule identifies and generates joins between the RDDs X and Y, instead of nested flatMaps, when $\text{vars}(e_1) \subseteq \text{vars}(p_1)$ and $\text{vars}(e_2) \subseteq \text{vars}(p_2)$, where function 'vars' returns the free variables in a pattern or expression:

$$\begin{aligned}&\text{rdd}[\ e \mid \overline{q_1},\ p_1 \leftarrow X,\ \overline{q_2},\ p_2 \leftarrow Y,\ \overline{q_3},\ e_1 == e_2,\ \overline{q_4}\ ]\\ &\quad=\text{rdd}[\ e \mid \overline{q_1},\ (\_,(p_1,p_2)) \leftarrow Z, \overline{q_2},\ \overline{q_3},\ \overline{q_4}\ ],\end{aligned}\tag{14}$$

where $Z = X.\text{map}(\lambda p_1.\,(e_1,p_1)).\text{join}(Y.\text{map}(\lambda p_2.\,(e_2,p_2)))$.

One way to represent arrays in a distributed setting is to store them as coordinate arrays, similar to the array representation used in Section 2. For instance, a matrix can be defined in Spark as an RDD of type RDD[((Long,Long),Double)], while matrix multiplication of two RDD matrices A and B, which was defined in (9), will be translated to the following program using Rules (14) and (13):

```
A.map{ case ((i,k),a) ⇒ (k,((i,k),a)) }
  .join( B.map{ case ((kk,j),b) ⇒ (kk,((kk,j),b)) } )
  .map{ case (_,(((i,k),a),((kk,j),b))) ⇒ ((i,j),a*b) }
  .reduceByKey(_+_).
```

Although correct, this Spark program has a high cost: it shuffles the matrices A and B across the compute nodes to perform the join, and then it shuffles all the products $A_{ik} * B_{kj}$ to perform the reduceByKey. Since data shuffling is the main cost factor for a distributed program, instead of fully sparse matrices in the coordinate format, we want to use a more compact representation for matrices by partitioning a matrix into tiles, which are unboxed arrays of type Array[Double] in which indices are calculated, not stored. The sparse matrix representation, on the other hand, is preferable when both dimensions of the matrix are large and the matrix is very sparse.

5 TRANSLATING BLOCK ARRAY QUERIES

A more effective way of representing an array in a distributed setting is to encode it as a distributed bag of non-overlapping blocks, where each block is a fix-sized chunk of the distributed array. A block is the unit of data distribution. Our goal in this section is to translate RDD comprehensions over block arrays to Spark's distributed data-parallel programs whose functional parameters will process the blocks very efficiently using multicore parallelism and array indexing. Given that data partitioning is necessary for data-parallel distributed processing, blocks are a natural way to partition large arrays and at the same time to minimize space overhead, compared to fully sparse arrays in coordinate format, in which the indices are stored along with a matrix element. This small space footprint translates to less data to shuffle across nodes and faster time to process each partition. Spark actually uses thread-level parallelism at each compute node to process the elements of each RDD partition in parallel using multicore parallelism, but the unit of parallelism for a block array is the entire block, which is likely to be one block for each compute node. Consequently, in addition to generating distributed operations from array comprehensions on block arrays, our goal is to process the data inside blocks using multicore parallelism.

In this paper, we focus on tiled matrices but our work can be easily extended to handle other block arrays too. We represent a tiled matrix using the following Scala class:

```
case class Tiled[T] ( rows: Long, cols: Long,
                      tiles: RDD[((Long,Long),Array[T])] ),
```

where rows is the number of rows, cols is the number of columns, and tiles is an RDD of fix-sized square tiles, where each tile ((i,j),A) has coordinates i and j and values stored in the array A. The array A has a fixed size N*N, for some constant N which is the same for all tiles. The coordinates i and j of a tile are unique, that is, tiles is an association list. A matrix element with indices ij is stored in the tile that has coordinates $(i/N, j/N)$ at the location $(i\%N) * N + (j\%N)$ inside the tile. The tile sparsifier is as follows:

```
def sparsify[T] ( S: Tiled[T] ): List[((Long,Long),T)]
  = [ ( ( ii*N+i, jj*N+j ), a(i*N+j) )
    | ((ii,jj),a) ← S.tiles,
      i ← 0 until N, j ← 0 until N ],
```

where ii and jj are the tile coordinates, and i and j are indices within a tile. The tiled builder uses the rdd and the array builders:

```
def tiled[T] ( n: Long, m: Long )
             ( L: List[((Long,Long),T)] ): Tiled[T]
  = Tiled( n, m, rdd[ ( (ii, jj), array(N*N)(w) )
                    | ((i,j),v) ← L, let ii = i/N, let jj = j/N,
                      let w = ( (i%N)*N+(j%N), v ),
                      group by (ii,jj) ] ),
```

where the group-by collects all tile elements into an array. The group-by comprehension is in an RDD comprehension, which means that it will be translated to a groupByKey operation in Spark, which requires data shuffling across the compute nodes. However, in some cases, this group-by qualifier can be eliminated, as in the case of a map over a matrix. Such an optimization is actually a general optimization over comprehensions with a group-by. A group-by qualifier in a comprehension can be eliminated if the group-by key is unique, that is, when the group-by function is injective. Although it is in general undecidable to prove whether a group-by key is unique, it is easy to do so for special cases, such as when the group-by key consists of array indices that are bound through an array traversal. For an array or map A, and the pattern variables v_i in $\overline{q_1}$ or $\overline{q_2}$, we have:

$$[\, e \mid \overline{q_1},\ (k,v) \leftarrow A,\ \overline{q_2},\ \textbf{group by}\ k\,] \\ = [\, e \mid \overline{q_1},\ (k,v) \leftarrow A,\ \textbf{let}\ \overline{v} = \text{unzip}([\,\overline{v} \mid \overline{q_2}\,])\,], \tag{15}$$

since the generator $(k,v) \leftarrow A$ for an array or map A indicates that k is unique. That is, this group-by is removed and every pattern variable v_i in $\overline{q_1}$ or $\overline{q_2}$ is lifted to a bag that contains all its values in the group. A similar rule exists for a generator $k \leftarrow e_1$ **until** e_2, since every value of k is unique.

Although correct, unfolding and normalizing tiled array comprehensions based on the tiled sparsifier and builder do not always result to optimal translations. In the rest of this section, we present special rules to translate tiled array comprehensions to efficient Spark programs.

5.1 Queries that Preserve Tiling

Consider the block comprehension without a group-by:

$$\text{tiled}(\overline{d})[\,(key, e) \mid \overline{q}\,] \tag{16}$$

where $\overline{d}$ is the tile dimensions (e.g., (m,n) for a matrix), key is the tile indices (e.g., (i,j) for a matrix) and e is the associated value. Let $(\overline{k_i}, v_i) \leftarrow X_i$ be a generator over a tiled array X_i in $\overline{q}$, where $\overline{k_i}$ is a tuple of index variables, such as $((i,j),v) \leftarrow X$ for a tiled matrix X. We say that this tiled comprehension *preserves tiling* if key is a tuple $\overline{w}$ that consists of variables that are defined in the tuples $\overline{k_i}$. The rest of the variables in the tuples $\overline{k_i}$ (not in $\overline{w}$) must be related to the variables in $\overline{w}$ with equality predicates in $\overline{q}$, so that the index of the constructed array is unique. For example, matrix addition and matrix diagonal preserve tiling:

```
tiled(n,m)[ ((i,j),a+b) | ((i,j),a) ← A, ((ii,jj),b) ← B,
                          ii == i, jj == j ],
tiled(n)[ (i,a) | ((i,j),a) ← A, i == j ].
```

A comprehension that preserves tiling is translated to an RDD comprehension that does not need a group-by to shuffle tiles. More specifically, the comprehension (16) is translated to:

$$\text{Tiled}(\overline{d}, \text{rdd}[\,(\overline{w}, \text{array}(N*N)[\,(\overline{w}, e) \mid \overline{q_2}\,]) \mid \overline{q_1}\,]), \tag{17}$$

where the qualifiers in $\overline{q_1}$ are those from $\overline{q}$ that do not refer to the tile values and each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ has been modified to be $(\overline{k_i}, \_v_i) \leftarrow X_i$.tiles (given the pattern variable v_i bound to an array value, $\_v_i$ is bound to the entire tile in $\overline{q_1}$). The qualifier list $\overline{q_2}$ is equal to $\overline{q}$ but each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ in $\overline{q}$ has been modified to be $(\overline{k_i}, v_i) \leftarrow \_v_i$. For example, matrix addition:

```
tiled(n,m)[ ((i,j),a+b) | ((i,j),a) ← A, ((ii,jj),b) ← B,
                          ii == i, jj == j ]
```

is translated to:

```
Tiled( n, m, rdd[ ( (i,j), V(_a,_b) )
                | ((i,j),_a) ← A.tiles, ((ii,jj),_b) ← B.tiles,
                  ii == i, jj == j ] ),
```

where V(_a,_b) is:

```
array(N*N)[ ((i,j),a+b) | ((i,j),a) ← _a, ((ii,jj),b) ← _b,
                          ii == i, jj == j ],
```

which is a regular array comprehension in which the tiles _a and _b will be lifted using the following array sparsifier for a tile A:

```
[ ((i,j),A(i*N+j)) | i ← 0 until N, j ← 0 until N ].
```

Based on the translation of RDD and array comprehensions, matrix addition is translated to the following Spark code:

```
Tiled( n, m, A.tiles.join(B.tiles)
         .map{ case ((ii,jj),(_a,_b)) ⇒ ((ii,jj),V(_a,_b)) } ).
```

After optimizations similar to those for matrix addition for regular matrices, we get the following code for V(_a,_b):

```
{ val V = Array.ofDim[Double](N*N);
  [ V((i%N)*N+(j%N)) = _a( (i%N)*N+(j%N) )
                     + _b( (i%N)*N+(j%N) )
  | i ← (0 until N).par, /* multicore parallelism */
    j ← 0 until N ];
  V }.
```

5.2 Queries that do not Preserve Tiling

For those array comprehensions that do not have a group-by and do not preserve tiling, we need to shuffle only the relevant tiles to the appropriate reducers. The array indices $\overline{w}$ in the result of the tiled comprehension

$$\text{tiled}(\overline{d})[\,(\overline{w}, e) \mid \overline{q}\,] \tag{18}$$

may now be arbitrary expressions that depend on the indices of the tiled generators in $\overline{q}$.

Let $f(\overline{k})$ be a term that depends on the array indices $\overline{k} = k_1, \ldots, k_m$ from the tiled generators in $\overline{q}$. The tiles accessed from the tiled generators $\overline{q}$ will have tile coordinates $\overline{K}$, where $K_i = k_i/N$. Given the tile coordinates $\overline{K}$ of the input tiles, the tile coordinate returned by $f(\overline{k})$ would be equal to $f(K_1 * N + j_1, \ldots, K_m * N + j_m)/N$, for $j_i \in [0, N)$ (the tile dimensions). We define, $\mathcal{I}_f(\overline{K})$ to be the set of all such tile coordinates:

$$\text{set}[\, f(K_1 * N + j_1, \ldots, K_m * N + j_m)/N \mid j_1 \leftarrow 0 \,\textbf{until}\, N, \ \ldots, \ j_m \leftarrow 0 \,\textbf{until}\, N\,],$$

where set(s) returns the distinct values of s. For example, if $f(k) = k+1$, then $\mathcal{I}_f(K) = \text{set}[\, f(K*N+j)/N \mid j \leftarrow 0 \,\textbf{until}\, N\,]$, which is equal to the set $\{K, K+1\}$ since $f(K * N + j)/N = (K*N+j+1)/N = K+(j+1)/N$. On the other hand, if $f(k) = k$, then $\mathcal{I}_f(K) = \{K\}$.

Based on this definition, comprehension (18) can be transformed to:

$$\text{Tiled}(\overline{d}, \text{rdd}[\,(\overline{K}, V) \mid \overline{q_1}, K_1 \leftarrow \mathcal{I}_{w_1}(\overline{k}), \ \ldots, \ K_m \leftarrow \mathcal{I}_{w_m}(\overline{k}), \ \textbf{group by}\, \overline{K}\,]), \tag{19}$$

where V is:

$$\text{array}[\,(\overline{w}, e) \mid \overline{q_2}, \ \textstyle\bigwedge_i \overline{K_i} == w_i(\overline{\_k} * N + \overline{k})/N\,].$$

The qualifiers in $\overline{q_1}$ are those from $\overline{q}$ that do not refer to the tile values and each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ has been transformed to the two qualifiers $(\overline{k_i}, \_v_i) \leftarrow X_i$.tiles and **let** $\_\_v_i = (\overline{k_i}, \_v_i)$ (given the pattern variable v_i bound to an array value, $\_v_i$ is bound to the entire tile in $\overline{q_1}$, while $\_\_v_i$ is bound to the index-tile pair). The group-by operation shuffles all the required tiles to the reducers to compute the resulting tiles. The qualifier list $\overline{q_2}$ in V is equal to $\overline{q}$ but each tiled generator $(\overline{k_i}, v_i) \leftarrow X_i$ in $\overline{q}$ has been transformed to the two qualifiers $(\overline{\_k_i}, \_v_i) \leftarrow \_\_v_i$ and $(\overline{k_i}, v_i) \leftarrow \_v_i$. The guards $\overline{K_i} == w_i(\overline{\_k} * N + \overline{k})/N$ selects only the proper tiles from all those shuffled by group-by.

For example, the following comprehension rotates the rows so that the first row is moved to the second, the second to third, etc, and the last to the first:

```
tiled(n,m)[ ( ( (i+1)%m, j ), v ) | ((i,j),v) ← X ].
```

The shuffled tiles for a tile in X with coordinates (i,j) have row coordinates from set[(i*N+_i+1)%m/N | _i ← 0 **until** N], which will be evaluated to List(i,i+1) for i < n/N and to List(i,0) for i = n/N, and column coordinates from List(j). That is, for each tile with coordinates (i,j) such that i < n/N, we require two tiles: the tile itself and its row successor with coordinates (i+1,j). Hence, the row rotation is translated to:

```
Tiled(n,m,rdd[ ( (K1,K2), V )
             | ((i,j),_a) ← X.tiles, let __a = ((i,j),_a),
               K1 ← set[ (i*N+_i+1)%m/N | _i ← 0 until N ],
               K2 ← List(j),
               group by (K1,K2) ] ),
```

which is translated to a Spark groupByKey operation. V is:

```
array(N*N)[ (((i+1)%m,j),a) | ((_i,_j),_a) ← __a, ((i,j),a) ← _a,
            K1 == (_i*N+i+1)%m/N, K2 == (_j*N+j)/N ] ),
```

which is translated to an efficient array-based program over the list of tiles __a.

5.3 Queries with a Group-By

Tile comprehensions with a group-by do not preserve tiling. They can be translated in a way similar to that for comprehension (18), but we can do better by using the RDD operation reduceByKey instead of the RDD operation groupByKey, because a group-by in a group-by comprehension is often followed by aggregation. A reduceByKey($\oplus$) operation, for some monoid $\oplus$, is equivalent to a groupByKey followed by reduction of each group using $\oplus$. Although functionally equivalent, reduceByKey is far more efficient than groupByKey in a distributed setting because it partially reduces the groups locally before they are shuffled to the reducers for the final reduction, resulting to less data shuffling.

Before we describe the general translation scheme, we explain the key idea using the example of matrix multiplication of the tiled matrices A and B:

```
tiled(n,m)[ ((i,j),+/v)) | ((i,k),a) ← A, ((kk,j),a) ← B,
                           kk == k, let v = a*b,
                           group by (i,j) ].
```

We want to translate it to the reduceByKey operation:

```
Tiled( n, m, rdd[ ((i,j),V(_a,_b))
                | ((i,k),_a) ← A.tiles, ((kk,j),_a) ← B.tiles,
                  kk == k ].reduceByKey(⊕) ),
```

where the tile V(_a,_b) is the tile _a multiplied by _b:

```
array(N*N)[ ((i,j),+/v)) | ((i,k),a) ← _a, ((kk,j),a) ← _b,
                           kk == k, let v = a*b,
                           group by (i,j) ]
```

and the monoid $\oplus$ over the tiles _x and _y adds the tiles pairwise:

```
_x⊕_y = array(N*N)[ ((i,j),x+y) | ((i,j),x) ← _x, ((ii,jj),y) ← _y,
                                  ii == i, jj == j ].
```

That is, the rdd comprehension calculates the partial sum of products of all matching tiles and the reduceByKey calculates the final sums by adding the tiles pairwise. Then, after translating the rdd and array comprehensions, we will get:

```
Tiled(n,m,A.tiles.map{ case ((i,k),_a) ⇒ (k,((i,k),_a)) }
     .join( B.tiles.map{ case ((kk,j),_a) ⇒ (kk,((kk,j),_a)) } )
     .map{ case (_,(((i,k),_a),((kk,j),_a))) ⇒ ((i,j),V(_a,_b)) }
     .reduceByKey(⊕))
```

where V(_a,_b) is translated to efficient code that multiplies the tiles _a and _b:

```
{ val V = Array.ofDim[Double](N*N);
  for { i ← 0 until N; j ← 0 until N; k ← 0 until N }
    V(i*N+j) += _a(i*N+k)*_b(k*N+j);
  V },
```

and _x⊕_y is pairwise addition:

```
{ val V = Array.ofDim[Double](N*N);
  for { i ← 0 until N; j ← 0 until N }
    V(i*N+j) = _x(i*N+j)+_y(i*N+j);
  V }.
```

To generate these reduceByKey calls, we consider the group-by qualifier in combination with the aggregations in the comprehension. Recall that, based on the discussion in Section 3 and on Equation (12), any tiled comprehension with a group-by can be put into the following form:

$$\begin{aligned}&\text{tiled}(n,m)[\,(p,e) \mid \overline{q},\ \textbf{group by}\ p,\ \overline{q'}\,]\\ &\quad= \text{tiled}(n,m)[\,(p,z) \mid \overline{q},\ \textbf{group by}\ p,\ z \leftarrow [\,e \mid \overline{q'}\,]\,]\\ &\quad= \text{tiled}(n,m)[\,(p,z) \mid \overline{q},\ \textbf{group by}\ p,\\ &\qquad z \leftarrow f(\oplus_1/w_1.\text{map}(g_1),\ldots,\oplus_m/w_m.\text{map}(g_m))\,],\end{aligned}$$

for some variables w_i lifted by group-by, some monoids $\oplus_i$, and some functions g_i and f. That is, we abstract all reductions $\oplus_i/w_i.\text{map}(g_i)$ from the term $[\,e \mid \overline{q'}\,]$. Notice that, here, the key of the tiled comprehension is equal to the group-by key, p. Then, the group-by comprehension can be translated to the following reduceByKey operation:

$$\begin{aligned}&\text{Tiled}(n,m,\text{rdd}[\,(p,(\text{array}(N*N)[\,g_1(w_1) \mid \overline{q_2}\,],\ldots,\\ &\qquad\qquad \text{array}(N*N)[\,g_m(w_m) \mid \overline{q_2}\,]))\\ &\qquad\qquad \mid \overline{q_1}\,]\\ &\qquad .\text{reduceByKey}(\otimes').\text{mapValues}(f'))\end{aligned}$$

Like in (17), the qualifiers in $\overline{q_1}$ are those from $\overline{q}$ that do not refer to the tile values and each tiled generator $(\overline{k_i},v_i) \leftarrow X_i$ has been modified to be $(\overline{k_i},\_v_i) \leftarrow X_i$.tiles. The qualifier list $\overline{q_2}$ is equal to $\overline{q}$ but each tiled generator $(\overline{k_i},v_i) \leftarrow X_i$ in $\overline{q}$ has been modified to be $(\overline{k_i},v_i) \leftarrow \_v_i$. The monoid $\otimes'$ is:

$$(x_1,\ldots,x_m) \otimes' (y_1,\ldots,y_m) \;=\; (x_1 \oplus'_1 y_1,\ldots,x_m \oplus'_m y_m)$$

where $x_k \oplus'_k y_k$ applies $\oplus_k$ to the tile elements pairwise:

$$\begin{aligned}&\text{array}(N*N)[\,a \oplus_k b \mid ((i,j),a) \leftarrow x_k,\ ((ii,jj),b) \leftarrow y_k,\\ &\qquad\qquad ii == i,\ jj == j\,]\end{aligned}$$

Finally, $f'(x_1,\ldots,x_m)$ is:

$$\begin{aligned}&\text{array}(N*N)[\,f(a_1,\ldots,a_m)\\ &\qquad\qquad \mid ((i_1,j_1),a_1) \leftarrow x_1,\ldots,\ ((i_m,j_m),a_m) \leftarrow x_m,\\ &\qquad\qquad i_1 == \cdots == i_m,\ j_1 == \cdots == j_m\,]\end{aligned}$$

For the matrix multiplication example, there is only one reduction with $\oplus_1 = +$ and f is identity, which means that mapValues(f') is identity too.

5.4 Using a Group-By-Join

There is a special class of tiled comprehensions with a group-by that can be translated to more efficient code. Consider the following comprehension:

```
tiled(n,m)[ (k,⊕/c) | ((i,j),a) ← A, ((ii,jj),b) ← B,
                      kx(i,j) == ky(ii,jj),  let c = h(a,b),
                      group by k: ( gx(i,j), gy(ii,jj) ) ],
```

for some arbitrary term functions kx, ky, gx, gy, and h, and some aggregation ⊕. Notice that the key k of the output matrix is the group-by key, which must be a pair where one component depends on i,j and the other on ii,jj only. This is called a group-by-join because it is a join between A and B, followed by a group-by with aggregation. Matrix multiplication, defined in (9), is an example of a group-by-join. Many group-by comprehensions can be put in the group-by-join form by combining generators in pairs forming joins until we are left with two generators and a group-by. A group-by-join can be evaluated very efficiently by joining each row of tiles from A with each column of tiles from B, which requires that we replicate every tile from A and B. When applied to the matrix multiplication, this algorithm is equivalent to the block matrix multiplication implemented using the SUMMA algorithm [14]. The group-by-join is translated to the following Spark RDD code:

```
Tiled( n, m, rdd[ (k,V) | (k,(__a,__b)) ← As.cogroup(Bs) ] ),
```

where As, Bs, and V are

```
As = A.tiles.flatMap{ case ((i,j),a)
     ⇒ (0 until B.cols/N).map(k ⇒ ((gx(i,j),k),(kx(i,j),a))) }
Bs = B.tiles.flatMap{ case ((ii,jj),b)
     ⇒ (0 until A.rows/N).map(k ⇒ ((k,gy(ii,jj)),(ky(ii,jj),b))) }
V = array(N*N)[ (k,⊕/c)
          | (k1,_a) ← __a, (k2,_b) ← __b, k2 == k1,
            ((i,j),a) ←_ a, ((ii,jj),b) ← _b, kx(i,j) == ky(ii,jj),
            let c = h(a,b), group by k: (gx(i,j),gy(ii,jj)) ].
```

That is, the tiles in A are replicated B.cols/N times and the tiles in B are replicated A.rows/N times.

6 PERFORMANCE EVALUATION

Our system, SAC, has been implemented using Scala's compile-time reflection and macros. Our code generator uses the Scala typechecker to infer the types of the generator domains to select the appropriate sparsifiers based on these types. It translates array comprehensions to Scala code that calls Spark RDD operations whose functional arguments use the Scala's Parallel Collections library [20] for multicore parallelism. The produced Scala code is embedded in the rest of the Scala code generated at compile-time. The source code of our system is available on GitHub at https://github.com/fegaras/array.

We have evaluated the performance of our system relative to the Spark MLlib.linalg library [5]. MLlib uses the linear algebra package Breeze, but in our experiments, instead of using a native Breeze library implementation, such as OpenBLAS, we used the pure JVM implementation of this library. Although there are many linear algebra libraries that support distributed tiled arrays, MLlib is the closest to our work since it is built on top of Spark and has a Scala API.

The platform used for these evaluations was a small cluster of 4 nodes, where each node has one Xeon E5-2680v3 at 2.5GHz, with 24 cores, 128GB RAM, and 320GB SSD. For our experiments,

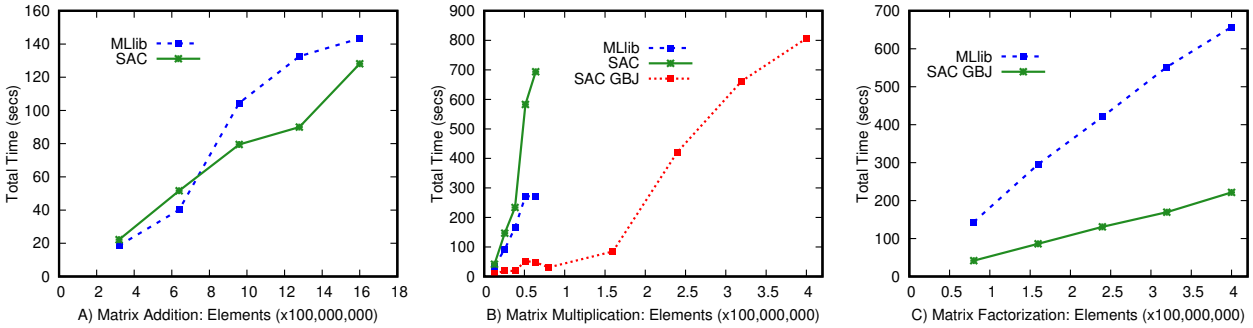


Figure 4: Performance evaluation of matrix addition, multiplication, and factorization of tiled matrices on Apache Spark

we used Apache Spark 3.0.0 running on Apache Hadoop 2.7.0. Each Spark executor was configured to have 11 cores and 60GB RAM. Consequently, there were 2 executors per node, giving a total of 8 executors. Each program was evaluated over 5 datasets and each evaluation was repeated 4 times, so each data point in the plots in Figure 4 represents the mean value of these 4 evaluations.

The matrices used in our experiments were tiled matrices where each tile had size 1000*1000. We implemented these distributed tiled matrices in MLlib.linalg as instances of BlockMatrix, where each tile was an instance of DenseMatrix. The matrices used for addition and multiplication were pairs of square matrices of the same size filled with random values between 0.0 and 10.0. The largest matrices used in matrix addition had 40000×40000 elements and size 12GB each, while those used in matrix multiplication had 20000×20000 elements and size 3GB each. Matrix multiplication was translated in two different ways in SAC: as a join followed by a group-by, and using a special group-by join (see Section 5.4 for a discussion). The results are shown in Figure 4.A and B. We can see that, for matrix addition, SAC performs a bit faster than MLlib. For matrix multiplication though, SAC (that uses a join followed by a group-by) is up to 3 times slower than MLlib, while MLib is up to 6 times slower than SAC GBJ (that uses the special group-by join).

The third program to evaluate was one iteration of matrix factorization using gradient descent [16]. The goal of this computation is to split a matrix R of dimension $n \times m$ into two low-rank matrices P and Q of dimensions $n \times k$ and $m \times k$, for small k, such that the error between the predicted and the original rating matrix $R - P \times Q^T$ is below some threshold, where $P \times Q^T$ is the matrix multiplication of P with the transpose of Q and '$-$' is cell-wise matrix subtraction. Matrix factorization can be implemented by repeatedly applying the following operations:

$$\begin{aligned} E &\leftarrow R - P \times Q^T \\ P &\leftarrow P + \gamma(2E \times Q - \lambda P) \\ Q &\leftarrow Q + \gamma(2E^T \times P - \lambda Q) \end{aligned}$$

where γ is the learning rate and λ is the normalization factor used in avoiding overfitting. For our experiments, we used $\gamma = 0.002$ and $\lambda = 0.02$. The matrix to be factorized, R, was a square sparse matrix $n*n$ with random integer values between 0 and 5, in which only the 10% of the elements were non-zero. The dimension k was set to 1000. The derived matrices P and Q had dimension $n * 1000$ and were initialized with random values between 0.0 and 1.0. The largest matrix R used had 20000×20000 elements and size 3GB. The results are shown in Figure 4.C. We can see that SAC (using GBJ) is up to three times faster than MLlib.

7 RELATED WORK

Many array-processing systems use special storage techniques, such as regular tiling, to achieve better performance on certain array computations. TileDB [19] is an array data storage management system that performs complex analytics on scientific data. It organizes array elements into ordered collections called fragments, where each fragment is dense or sparse, and groups contiguous array elements into data tiles of fixed capacity. Unlike our work, the focus of TileDB is on the I/O optimization of array operations by using small block updates to update the array stores. SciDB [22] is a large-scale data management system for scientific analysis based on an array data model with implicit ordering. The SciDB storage manager decomposes arrays into a number of equal sized and potentially overlapping chunks, in a way that allows parallel and pipeline processing of array data. Like SciDB, ArrayStore [21] stores arrays into chunks, which are typically the size of a storage block. One of their most effective storage method is a two-level chunking strategy with regular chunks and regular tiles. SciHadoop [7] is a Hadoop plugin that allows scientists to specify logical queries over arrays stored in the NetCDF file format. Their chunking strategy, which is called the Baseline partitioning strategy, subdivides the logical input into a set of partitions (sub-arrays), one for each physical block of the input file. SciHive [15] is a scalable array-based query system that enables scientists to process raw array datasets in parallel with a SQL-like query language. SciHive maps array datasets in NetCDF files to Hive tables and executes queries via Map-Reduce. Based on the mapping of array variables to Hive tables, SQL-like queries on arrays are translated to HiveQL queries on tables and then optimized by the Hive query optimizer. SciMATE [25] extends the Map-Reduce API to support the processing of the NetCDF and HDF5 scientific formats, in addition to flat-files. SciMATE supports various optimizations specific to scientific applications by selecting a small number of attributes used by an application and perform data partition based on these attributes. TensorFlow [1] is a dataflow language for machine learning that supports data parallelism on multi-core machines and GPUs but has limited support for distributed computing. Linalg [26] (now part of Spark's MLlib library) is a distributed linear algebra and optimization library that runs on Spark. It consists of fast and scalable implementations of standard matrix computations for common linear algebra operations, such as matrix multiplication and factorization. One of its distributed matrix representations, BlockMatrix, treats the matrix as dense blocks of data, where each block is small enough to fit in memory on a single machine. Linalg allows matrix computations to be pushed from the JVM

down to hardware via the Basic Linear Algebra Subprograms (BLAS) interface. SystemML [6] is a machine learning (ML) library built on top of Spark. It supports a high-level specification of ML algorithms that simplifies the development and deployment of ML algorithms by separating algorithm semantics from underlying data representations and runtime execution plans. Distributed matrices in SystemML are partitioned into fixed size blocks, called Binary Block Matrices. Although many of these systems support block matrices, their runtime systems are based on a library of build-in, hand-optimized linear algebra operations, which is hard to extend with new storage structures and algorithms. Furthermore, many of these systems lack a comprehensive framework for automatic inter-operator optimization, such as finding the best way to form the product of several matrices. Like these systems, our framework separates specification from implementation, but, unlike these systems, our system supports ad-hoc operations on array collections, rather than a library of build-in array operations, is extensible with customized storage structures, and uses relational-style optimizations to optimize array programs with multiple operations.

There has also been some recent work on combining linear algebra with relational algebra to let programmers implement ML algorithms on relational database systems [2, 17, 18]. The work by Luo *et al.* [18] adds a new attribute type to relational schemas to capture arrays that can fit in memory and extends SQL with array operators. Although their system evaluates SQL queries in Map-Reduce, the arrays are not fully distributed. Instead, large matrices must be split into multiple rows as indexed tiles while the programmer is expected to write SQL code to implement matrix operations by correlating these tiles using array operators in SQL. That is, SQL queries on distributed arrays are customizable but the array operators used in correlating tiles are build-in from a library. However, even if these tile operations were customizable, this system would differ from ours since it does not separate specification from implementation, thus making hard to change the array storage, and requires programmers to write explicit code to correlate tiles.

8 CONCLUSION AND FUTURE WORK

Our performance results show that SAC can be as efficient as a highly optimized array library when applied to certain distributed operations on tiled arrays. The same layered approach can also be used for translating comprehensions on other types of array storage, such as on tiled arrays where each tile is stored in the compressed sparse column format. As future work, we plan to look at operations that are hard to express using comprehensions, such as inverting a matrix, which requires a special LU decomposition algorithm. We believe that such operations should be coded as black-box library functions in a high-performance array library, such as BLAS or LAPACK. Such operations would require special optimizations to fuse them with general array comprehensions and with each other. Furthermore, our framework cannot directly generate calls to a high-performance array library, such as BLAS or LAPACK, or to GPU libraries, such as CUDA or OpenGL, but it can be improved to recognize certain patterns in a comprehension that are translatable to such calls. Finally, we would like to investigate general methods to optimize storage, such as unboxing arrays where vectors of tuples are mapped to tuples of vectors, and to parallelize irregular structures using nested parallelism.

Acknowledgments: Our evaluations were performed at the XSEDE Comet cloud computing infrastructure at SDSC, www.xsede.org, supported by NSF.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[2] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. Towards linear algebra over normalized data. *Proceedings of the VLDB Endowment (PVLDB)*, 10(11):1214—1225, 2017.

[3] Apache Flink. http://flink.apache.org/, 2020.

[4] Apache Spark. http://spark.apache.org/, 2020.

[5] Apache Spark MLlib. https://spark.apache.org/mllib/, 2020.

[6] M. Boehm, M. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13):1425–1436, 2016.

[7] J. Buck, N. Watkins, J. Lefevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. A. Brandt. SciHadoop: Array-based Query Processing in Hadoop. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[9] J. J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. In *ACM Transactions on Mathematical Software*, 16(1):1—17, 1990.

[10] L. Fegaras. An Algebra for Distributed Big Data Analytics. *JFP*, special issue on Programming Languages for Big Data, volume 27, 2017.

[11] L. Fegaras. A Query Processing Framework for Large-Scale Scientific Data Analysis. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS)*, Springer, July 2018.

[12] L. Fegaras and M. H. Noor. Compile-Time Code Generation for Embedded Data-Intensive Query Languages. In *IEEE BigData Congress*, 2018.

[13] L. Fegaras and M. H. Noor. Translation of Array-Based Loops to Distributed Data-Parallel Programs. *Proceedings of the VLDB Endowment (PVLDB)*, 13(8): 1248-1260, 2020.

[14] R. A. Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. In *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[15] Y. Geng, X. Huang, M. Zhu, H. Ruan, and G. Yang. SciHive: Array-based query processing with HiveQL. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (Trustcom)*, 2013.

[16] Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems *IEEE Computer*, 42(8):30–37, August 2009.

[17] A. Kunft, A. Alexandrov, A. Katsifodimos, and V. Markl. Bridging the gap: towards optimization across linear and relational algebra. In *ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, pages 1–4, 2016.

[18] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 31(7):1224—1238, 2018.

[19] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The TileDB Array Data Storage Manager. *PVLDB*, 10(4):349–360, 2016.

[20] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A Generic Parallel Collection Framework. In *Euro-Par Parallel Processing*, 2011.

[21] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *ACM SIGMOD International Conference on Management of Data*, pages 253–264, 2011.

[22] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly. Efficient Iterative Processing in the SciDB Parallel Array Engine. In *27th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2015.

[23] P. Wadler. List Comprehensions. Chapter 7 in *The Implementation of Functional Programming Languages*, by S. Peyton Jones, Prentice Hall, 1987.

[24] P. Wadler and S. Peyton Jones. Comprehensive Comprehensions (Comprehensions with 'Order by' and 'Group by'). In *Haskell Symposium*, pages 61–72, 2007.

[25] Y. Wang, W. Jiang, and G. Agrawal. SciMATE: A Novel MapReduce-like Framework for Multiple Scientific Data Formats. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.

[26] R. B. Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, and M. Zaharia. Matrix Computations and Optimization in Apache Spark. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 31–38, 2016.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

Short Paper

Automated Machine Learning for Entity Matching Tasks

Matteo Paganelli, Francesco Del Buono, Francesco Guerra, Marco Pevarello, Maurizio Vincini
firstname.lastname@unimore.it
University of Modena and Reggio Emilia
Modena, Italy

ABSTRACT

The paper studies the application of automated machine learning approaches (AutoML) for addressing the problem of Entity Matching (EM). This would make the existing, highly effective, Machine Learning (ML) and Deep Learning based approaches for EM usable also by non-expert users, who do not have the expertise to train and tune such complex systems. Our experiments show that the direct application of AutoML systems to this scenario does not provide high quality results. To address this issue, we introduce a new component, the *EM adapter*, to be pipelined with standard AutoML systems, that preprocesses the EM datasets to make them usable by automated approaches. The experimental evaluation shows that our proposal obtains the same effectiveness as the state-of-the-art EM systems, but it does not require any skill on ML to tune it.

1 INTRODUCTION

Machine Learning (ML) has significantly advanced over the past ten years [1]. On one side, the research on Big Data let emerge new challenges and made available scenarios and datasets where to experiment and improve ML techniques. On the other side, the increase of computer processing power, thanks in particular to the use of graphic processing units, enabled ML approaches running in commodity hardware. This led to the development of new ML algorithms and their implementations through frameworks and libraries is extensive and growing [17]. Thus the ML technology moved from an R&D phase, for the exclusive use of specialized laboratories, to a mature phase where it can be adopted in business applications.

Mature technologies have to be easy to use for both non-experts and professionals. One of the main bottlenecks towards a large use of the ML technology is related to the configuration of the systems, where experts are typically needed to set the large number of hyper-parameters. Furthermore, the selection of the algorithm that best performs in a given ML task is based on an experimental evaluation in which the performances of competing approaches are compared. This requires a time-consuming and expensive iterative process in which multiple alternative solutions are tested until an optimal result is achieved.

To address these issues, automated machine learning (AutoML) tools have been proposed. These are user-friendly and easy-to-use systems that provide a unified interface for the automatic selection of the most appropriate ML model/algorithm for a given task and its automatic configuration. Some examples are Auto-WEKA [12], AutoSklearn [9], AutoGluon [8], Auto-Keras [11], H20 AutoML [10], and many other.

This paper analyzes the application of AutoML systems to Entity Matching (EM), i.e. the task of identifying which records in a dataset refer to the same real-world entity [5]. Applications addressing EM tasks are becoming mature technologies and AutoML can promote this maturation for three main reasons. First of all, the state of the art EM systems are based on complex and advanced neural architectures [2, 7, 13, 16, 25] which require a non-trivial configuration process performed by expert users. Therefore, AutoML can make the application of EM techniques possible also to less experienced users. Secondly, the implementation of an EM system based on ML is expensive since it is performed through a time-consuming process that requires the active involvement of domain experts for evaluating the model. Introducing a form of automation would reduce the cost and time for the model deployment. Finally, in business scenarios where annotating data for the training process is costly and the performance evaluation is a critical task, the results obtained with an AutoML system represent a low-cost baseline.

Our experiments showed that the direct application of AutoML systems to the EM task was not effective. One of the main reasons is that such systems have been designed to solve generic ML tasks. Therefore, they are ineffective in dealing with the peculiarities of EM, characterized by an "unusual" data format, where each record is a description of pairs of entities, and there is a high imbalance between the “match" and “non-match" classes to predict.

For this reason, we introduce in this paper a software component, the *EM adapter*, encoding the datasets used in the EM task into a numerical form that make them to be effectively processed by AutoML systems. The *EM adapters* rely on the most recent transformer architectures, such as BERT and variants [6, 14, 19, 24], whose out of the box application to the EM problem has proved to be particularly effective [2]. We show in our experiments that *EM adapters* pipelined with AutoML systems are able to ensure quality performance comparable with the one of EM approaches parametrized by ML experts, thus making possible the use of complex ML techniques to non-expert users.

Summarizing, the main contributions of our paper are:

- An analysis about the application of AutoML systems to the EM task;
- The design of *EM adapters*, components based on pre-trained transformer architectures, to be pipelined with AutoML systems for making them able to effectively perform the EM task;
- An extensive experimentation (including a comparison with other state of the art EM tools) to evaluate the effectiveness of our proposal in different scenarios (reduced training times and different types of data).

The rest of the paper is organized as follows. Section 2 introduces the state-of-the-art techniques developed in the AutoML and EM fields. Section 3 describes the main features of the *EM adapters*, the components we developed for making AutoML systems addressing EM tasks, and Section 4 presents the implementation we developed. Our approach has been experimented as described in Section 5. Finally, in Section 6 we sketched out some conclusions and future work.

2 RELATED WORK

Entity Matching. Despite the effort put by the research community for more than thirty years, EM is still an open challenge. Several techniques have been proposed: they range from rules-based approaches [18, 20, 22] to ML models, that conceive EM as a binary classification problem [4] applied on datasets whose records describe pairs of entities. Recently, Deep Learning (DL) approaches (DeepER [7] and DeepMatcher [16] are the first proposals) have proved to be very effective. They suffer from two main problems: 1) the need for a significant amount of labeled data for their training and 2) a non-trivial configuration. To address the first problem, approaches based on transfer learning and fine-tuning techniques relying on architectures pre-trained on large generalist corpus have been proposed [2, 13, 25]. [25] presents a transfer learning approach to EM leveraging on pre-trained models from large-scale, production knowledge bases. [13] applies a fine-tuning process based on a pre-trained Bert architecture [6] using a limited number of labeled data, whose performance is further improved through the exploitation of data augmentation techniques. An analogous approach is described in [2], in which a larger collection of transformer architectures (Bert, XLNet [24], DistilBERT [19] and RoBERTa [14]) are applied to the EM problem.

Although these systems address the problem of the need for a significant number of labeled data, they require specific skills in modifying, training, and configuring complex neural architectures. In this work, we aim to make an EM architecture accessible also to non-expert users and to generate low-cost baselines for EM tasks.

AutoML AutoML is a recent research topic and consists in the automatic identification of the most effective algorithms to make ML inference requiring minimal human intervention and exploiting a limited computational budget (e.g. in terms of use of CPU and RAM).

The first definition of this problem is given in [12], where AutoML is formalized as a *Combined Algorithm Selection and Hyperparameter optimization (CASH)* problem. The solution proposed in this work is Auto-WEKA: a system designed to automatically select and configure, using methods based on Bayesian optimization, the classification models available within the WEKA[1] ML library. Starting with this pioneering work, numerous AutoML systems have been developed. Among them, AutoSklearn [9] combines a meta-learning technique, to boost a Bayesian optimization, with an ensemble method applied to the models selected in the previous step. AutoGluon [8] uses ensembling and multi-layer stacking techniques to combine multiple models (such as LightGBM boosted trees, CatBoost boosted trees, Random Forests, Extremely Randomized Trees, and k-Nearest Neighbors). In the context of neural networks, the main exponent is Auto-Keras [11], which applies Bayesian optimization in the search for the most efficient neural architecture (the so-called *Neural Architecture Search - NAS*). Another promising AutoML system is H2OAutoML [10], which, in place of Bayesian optimization, uses a combination of fast random search with ensemble staking techniques. Finally, a comparison of state-of-the-art approaches is provided in [21].

To the best of our knowledge, no study has been proposed for the application of AutoML systems to the EM task.

[1]https://www.cs.waikato.ac.nz/~ml/weka/index.html

3 DESIGNING AUTOML FOR EM TASKS

AutoML systems could largely support the application of ML and DL approaches to EM tasks by selecting the best classification models for given datasets and providing the tuning of the parameters. Nevertheless, our experiments (partially reported in Section 5) showed that AutoML systems are not effective in this scenario.

To investigate the reasons for such behavior, we considered AutoML systems as black-box tools and we analyzed if there are peculiarities that make EM, conceived as ML task, a hard issue for these approaches. Firstly, we observed that the datasets used for representing EM tasks describe pairs of entities. The insight that an ML model can learn from them is mainly obtained by comparing the values assumed by pairs of attributes describing the same feature in different entities. Since an entity is described through several features, and pairs of attributes describing the same features have value distributions clearly close, selecting and tuning an ML model can be a complex task. Secondly, the classification problem is highly imbalanced, due to the very large number of unmatching entities with respect to the matching ones.

In this paper, we address the first issue by developing and experimenting with a component, the *EM adapter*, which provides an encoding of EM datasets that effectively supports the training of AutoML systems. We consider the second issue as future work that we intend to address designing data augmentation techniques for building a more balanced dataset to train the AutoML system.

The existing AutoML systems provide limited pre-processing and feature extraction capabilities. Our approach represents a first attempt towards the development of a new generation of AutoML systems able to operate in specific scenarios thanks to data transformation capabilities. The design of the *EM adapter* was based on a preliminary analysis of the main features implemented in the existing approaches. This allowed us to get insights into the design choices to adopt in the creation of the component.

The existing state-of-the-art approaches for solving EM tasks are typically based on neural architectures, and in particular on recurrent neural networks (RNNs), as DeepER and DeepMatcher. The RNN (e.g., a bi–directional recurring network made up of LSTM cells in DeepER) is trained to encode the pairs of entities, input of the network, into multi-dimensional vectors. The application of some similarity function to these vectors makes them a training dataset for a binary classifier. The systems based on these architectures are highly effective thus demonstrating their ability in managing the particular schemas adopted by datasets describing EM tasks.

Insight#1: RNNs are an effective means to represent in a single multi-dimensional vector the values of the attributes describing the same feature of pairs of entities.

Training models based on RNN requires large datasets, thus preventing their use in several business scenarios. To address this issue, approaches have been experimented RNNs trained on large “external" knowledge bases and applied to in-production systems via transfer learning techniques. Transfer learning has been successfully applied to several ML applications, and to tasks

related to the EM as in [25] where the problem was recognizing categories where entity features belong to.

Insight#2: External knowledge-bases can be effectively exploited in ML architectures addressing EM tasks to reduce the need for annotated data.

Transformers [23] have recently gained large popularity in the NLP field. They are deep neural networks that, once trained on a large corpus, are able to learn the semantics of words better than conventional word embedding techniques (e.g., Word2vec [15]). They have also been proved effective in EM tasks [2, 13]. In [13], for example, a fine-tuning process is applied to a pre-trained Bert transformer architecture. The dataset is completely denormalized and, in the resulting text fields, meta-tokens are inserted not to lose the semantics of the schema. In this way, the knowledge through which an ML algorithm learns to discriminate between pairs of matching/non-matching entities is no longer obtained by comparing the values assumed by pairs of attributes, but emerges from the analysis of the record as a whole. A broad experimentation of transformers approaches in the field of EM performed in [2] let emerge two main findings: 1) the transformers can be used for EM out of the box, without the need for a task-specific architecture, and 2) fine-tuning a transformer on an EM task takes relatively little time and requires no particularly capable hardware.

Insight#3: Transformers serialize in numerical vectors the fields of EM datasets describing pairs of entities by exploiting a highly-contextualized analysis of EM data. Their use in EM tasks is a good compromise between simplicity and performance, and their fine-tuning does not require high training times and expensive hardware.

4 IMPLEMENTING AUTOML FOR EM TASKS

The *EM adapter* is the component in charge of computing an encoding of a dataset representing EM to be effectively exploited by an AutoML system. Its functional architecture is based on the insights defined in Section 3 and consists of three main components: the *Tokenizer* which tokenizes the dataset records into one or more token sequences; the *Embedder* which transforms the token sequences into embeddings; and the *Combiner* which generates a single multi-dimensional vector from all embeddings associated the same dataset entry.

Tokenizer. This component transforms an entity pair $(e1, e2)$, described in the records of the EM dataset as a series of attributes $a_{11}, ..., a_{1M}, a_{21}, ..., a_{2M}$, where a_{ij} is the attribute j of the entity e_i, into one or more token sequences. The component defines how to combine the values of the records to obtain tokens that enable the automatic learning of deep relations between attributes describing the same features of different entities. There are three alternative ways for performing this operation: the unstructured, attribute-based, and hybrid tokenization mode. In the unstructured mode, all fields describing the entities are concatenated into a unique sentence and any reference to the dataset schema is lost. In the attribute-based mode, the dataset entry is tokenized at the attribute level and the values of the same attribute for the considered pair of entities are coupled. Through this kind of tokenization, the records in the EM datasets describing pairs of entities are broken down into multiple sub-pairs, one for each attribute. Finally, in the hybrid mode, partial and / or incremental concatenations of the attribute values are performed. A hybrid strategy can for example apply an attribute-level tokenization and then incrementally combine the sub-pairs so that the $i-th$ pair contains the values of the first i attributes, and the last one compares the entire original matching pair. The aforementioned hybrid technique and the attribute-based technique have been experimented in the paper.

Dataset	Type	Datasets	Size	% Match
S-DG	Structured	DBLP-GoogleScholar	28,707	18.63
S-DA		DBLP-ACM	12,363	17.96
S-AG		Amazon-Google	11,460	10.18
S-WA		Walmart-Amazon	10,242	9.39
S-BR		BeerAdvo-RateBeer	450	15.11
S-IA		iTunes-Amazon	539	24.49
S-FZ		Fodors-Zagats	946	11.63
T-AB	Textual	Abt-Buy	9,575	10.74
D-IA	Dirty	iTunes-Amazon	539	24.49
D-DA		DBLP-ACM	12,363	17.96
D-DG		DBLP-GoogleScholar	28,707	18.63
D-WA		Walmart-Amazon	10,242	9.39

Table 1: Magellan Benchmark

Regardless of the specific implementation, the result of this step consists of one or more token sequences for each entry of the original EM dataset.

Embedder. The goal of the *Embedder* is to encode a token sequence into a multi-dimensional vector, i.e., an embedding. The approach usually adopted in the literature starts from a pre-trained word embedding deep learning architecture which is experimented with some token sequences. The embeddings are generally then extracted from these architectures by averaging the last hidden layer for each token, but other techniques have been experimented (the concatenations of the last 4 hidden layers for each token in [6]). In the paper, we experimented five embedders: Bert, DistilBert (DBert), Albert, Roberta and XLNET.

The *Embedder* outputs an embedding for each input token sequence. Recall that, according to the tokenization strategy adopted, an entry in the dataset can generate from one to many embeddings.

Combiner. The embeddings generated from each entry in the EM dataset are summarized in a single multi-dimensional vector by the *Combiner*. The standard approach, experimented in the paper, for performing this task is to calculate an average embedding.

5 EXPERIMENTAL EVALUATION

This section aims to provide an experimental answer to three main questions: 1) How effective are standard AutoML systems in solving EM tasks (Section 5.1); 2) To which extent the performance of AutoML systems applied to the EM task benefit from data preprocessing techniques (Section 5.2); and 3) How much AutoML systems pipelined with the *EM adapters* outperform the current state-of-the-art EM models (Section 5.3).

Datasets. The experiments have been performed against the datasets provided by the Magellan library[2] which are considered as a standard benchmark for the evaluation of EM tasks. In Table 1 we summarize with some statistic measures the 12 datasets analyzed, reporting for each of them the total number of records representing matching entities (fourth column) and the percentage of records associated with a matching label (last column). Each dataset is divided into training, validation, and test sets which were created with 60-20-20 proportions. The code used in the experiments is available at *https://github.com/softlab-unimore/automl-for-em*.

[2]https://github.com/anhaidgroup/deepmatcher/blob/master/Datasets.md

	AutoSklearn		AutoGluon		H2OAutoML		DeepMatcher	
	F1	Training time (h)	F1	Training time (h)	F1	Training time (h)	F1	Training time (h)
S-DG	50.65	1.00	**77.85**	4.42	64.74	0.97	94.70	8.50
S-DA	92.79	1.00	**97.62**	2.06	92.51	0.94	98.40	8.50
S-AG	**44.10**	1.00	23.28	1.57	36.88	0.96	69.30	1.50
S-WA	29.28	1.00	19.12	2.48	**31.07**	0.94	66.90	2.17
S-BR	40.00	1.00	0.00	0.07	**43.24**	0.74	72.70	0.08
S-IA	53.33	1.00	50.00	0.13	**59.09**	0.87	88.00	0.25
S-FZ	**100.00**	1.00	71.11	0.10	61.90	0.86	100	0.17
T-AB	26.47	1.00	11.41	1.63	**27.36**	0.94	62.80	3.50
D-IA	**64.00**	1.00	60.87	0.15	62.75	0.87	74.50	0.17
D-DA	54.74	1.00	**89.44**	3.07	67.92	0.94	98.10	4.00
D-DG	46.79	1.00	**69.05**	4.96	43.01	0.97	93.80	8.50
D-WA	25.75	1.00	14.12	2.10	**26.31**	0.97	46.00	2.50

Table 2: Effectiveness of AutoML systems in EM tasks.

5.1 AutoML systems solving EM tasks

The experiment shows the application of standards AutoML systems in solving EM tasks. We selected three well-known approaches, AutoSklearn, AutoGluon, and H2OAutoML. We evaluated their performance as binary classifiers against the chosen datasets. The AutoML systems have been experimented with their default configuration, no parameter tuning has been performed. Only the AutoSklearn system required the application of a data pre-processing step to transform categorical features (which are not managed by this system) into numerical values. We performed this operation via a standard Word2Vec embedding, where the average Word2Vec embedding for each token of non-numeric attributes has been computed and concatenated. The results of this experiment are shown in Table 2, reporting the F1 score and the training time of each AutoML system for each dataset. The last column in the Table shows the scores achieved by DeepMatcher, in the *Hybrid* configuration, which we consider as a baseline[3].

Discussion. The AutoML systems are not effective in solving EM tasks as DeepMatcher. They perform close to DeepMatcher (and with an average F1 score greater than 75%) in two datasets only (S-DA and S-FZ). Even if there are small variations among the performance achieved, there is no AutoML system that clearly outperforms the others: the average F1 scores along all the datasets are close, ranging from 48.66% for AutoGluon, to 51.4% for H2OAutoML, and 52.33% for AutoSklearn. Nevertheless, the time required for training the models was largely varying. H2OAutoML took 1 hour maximum to finalize the training in all the datasets; AutoSklearn limits the training time to one hour by default; the time required for training AutoGluon was larger, more than four hours for the S-DG and D-DG datasets. We performed a further experiment, limiting the training time of AutoGluon to 1 hour. The quality of the results largely decreased and the average F1 score across all datasets dropped to 42.4%.

Lesson Learned. AutoML systems are not competitive when addressing binary classification problems associated with EM tasks.

5.2 AutoML systems pipelined with EM adapters

In this section, we evaluate the effectiveness of AutoML systems coupled with an *EM adapter*. For this reason, we experimented several adapters, obtained by combining possible implementations for their constituting modules, the tokenizer, the embedder and the combiner, introduced in Section 4. In the following, we report the effectiveness measured in adapters implementing attribute-based and hybrid tokenizers, where the best results are obtained. For each kind of tokenizer, five standard transformer architectures, namely Bert, DistilBert (DBert), Albert, RoBERTa and XLNET have been evaluated[4]. For sake of simplicity, we only consider combiners generating the embeddings from the last hidden layer for each token and averaging it with the layers referring to the other tokens. Tables 3 shows the results of this evaluation for AutoSklearn, AutoGluon and H2OAutoML respectively. The scores of the *EM adapters* were grouped according to the tokenization technique (attribute-based vs hybrid). The F1 scores in bold represent the best result per dataset obtained for the category of tokenizer considered. The values in bold and underlined represent the results of the most effective *EM adapter* for the considered dataset.

Discussion. We observe that (1) the *EM adapters* implementing hybrid tokenizers typically obtain the best performance (only in 3/12 datasets, i.e. S-DA, S-WA and S-FZ, the results are worse); (2) the Albert embedder achieves the best results (i.e., *EM adapter* implementing an Albert embedder is the best solution for 7/12 datasets considering the AutoSklearn system and 8/12 considering AutoGluon and H2OAutoML). Finally, Table 4 reports an overall evaluation of the impact of an *EM adapter* as a preprocessing component for an AutoML system in solving EM tasks. For each dataset and AutoML system, the F1 scores obtained in the absence and presence of an *EM adapter* are reported. The "No EM-Adapter" column is the reference column showing the F1 scores obtained by AutoML systems with no *EM adapter*. The remaining columns show the average F1 score (through the 5 transformer architectures considered) obtained by the adapters implementing attribute-based and hybrid tokenizers. Finally, for each AutoML system, the delta column shows the difference between the F1 score obtained with no *EM adapter* and the average of the two versions including the *EM adapters*. The experiments show that adapters significantly improve the effectiveness of AutoML systems in solving EM tasks in almost all datasets (the datasets S-FZ for AutoSklearn and S-DA for AutoGluon show an anomaly result). The average F1 score increases of 24.96%, 28.02% and 23.6% for AutoSklearn, AutoGluon and H2OAutoML respectively.

Lesson Learned. *EM adapters* largely improve the effectiveness of AutoML systems in addressing EM tasks. The experiments were not able to show a clear winner among the approaches tested. This is a positive result since it means that the AutoML technology can benefit from the application of *EM adapters*.

5.3 EM adapters pipelined with AutoML systems vs ad hoc solutions

The aim of this experiment is to evaluate if an AutoML system pipelined with an *EM adapter* can obtain competitive results with respect to other state-of-the-art EM models. For this evaluation, we consider an AutoML system with an *EM adapter* consisting of a hybrid tokenizer and an Albert embedder, whose combination provided the best performances in the previous experiments.

[3]We used the implementation available at https://github.com/anhaidgroup/deepmatcher

[4]No fine-tuning technique was applied in the experiments.

(a) EM-Adapter with AutoSklearn

	Attr-EM-Adapter					Hybrid-EM-Adapter				
	Bert	**DBert**	**Albert**	**Roberta**	**XLNET**	**Bert**	**DBert**	**Albert**	**Roberta**	**XLNET**
S-DG	92.70	90.49	**92.85**	92.11	91.65	**93.76**	91.92	93.56	92.59	92.77
S-DA	**98.10**	95.74	96.54	97.89	96.76	**96.90**	95.73	96.64	96.12	96.22
S-AG	62.11	59.59	**66.67**	60.07	51.38	66.67	62.60	**68.41**	61.25	58.74
S-WA	58.05	55.73	**67.18**	55.21	56.36	56.93	52.36	**62.17**	48.00	46.92
S-BR	66.67	68.75	**73.33**	70.97	66.67	64.52	78.79	**74.29**	**74.29**	68.97
S-IA	88.46	88.89	83.64	80.77	**90.20**	83.64	87.27	85.71	88.46	**96.30**
S-FZ	97.67	97.78	95.24	**100.00**	93.02	**97.67**	**97.67**	95.24	**97.67**	**97.67**
T-AB	58.44	57.08	**66.37**	56.68	54.65	66.97	58.23	**76.92**	66.36	61.01
D-IA	56.52	58.82	**67.92**	51.85	64.52	80.70	77.97	**91.23**	79.25	87.72
D-DA	92.90	91.58	**96.30**	91.71	93.20	**96.58**	95.41	96.36	95.20	96.00
D-DG	86.73	85.63	**90.29**	86.22	86.67	93.00	91.39	**93.12**	92.76	92.25
D-WA	39.82	45.71	**56.02**	39.17	37.99	51.38	51.10	**62.80**	46.23	44.23

(b) EM-Adapter with AutoGluon

	Attr-EM-Adapter					Hybrid-EM-Adapter				
	Bert	**DBert**	**Albert**	**Roberta**	**XLNET**	**Bert**	**DBert**	**Albert**	**Roberta**	**XLNET**
S-DG	93.35	91.88	**93.53**	92.19	92.76	94.30	92.83	**94.37**	93.25	93.56
S-DA	98.19	97.31	96.85	**98.31**	97.64	96.67	96.31	**96.75**	96.32	96.21
S-AG	57.78	50.14	**64.61**	54.04	45.39	58.06	58.55	**64.89**	53.40	55.83
S-WA	54.07	58.13	**67.04**	52.04	52.34	55.46	49.52	**64.67**	47.88	42.99
S-BR	64.29	66.67	**80.00**	63.64	64.29	64.29	64.29	**81.25**	71.43	75.86
S-IA	84.62	**88.46**	85.19	85.19	79.17	86.79	86.79	92.59	86.79	**98.18**
S-FZ	97.67	**100.00**	97.67	97.67	95.24	97.67	**100.00**	95.24	**100.00**	**100.00**
T-AB	58.49	58.72	**70.95**	56.55	56.76	60.18	64.22	**73.97**	63.69	59.71
D-IA	**69.39**	61.90	63.83	58.33	50.00	80.00	77.55	**87.27**	81.48	82.35
D-DA	93.00	93.12	**96.02**	91.05	92.97	96.54	95.58	**96.76**	96.00	95.88
D-DG	88.78	87.78	**91.73**	87.86	88.45	92.51	92.35	93.10	**93.16**	92.58
D-WA	32.47	44.52	**56.89**	33.79	33.55	51.55	49.35	**65.56**	43.29	39.13

(c) EM-Adapter with H2OAutoML

	Attr-EM-Adapter					Hybrid-EM-Adapter				
	Bert	**DBert**	**Albert**	**Roberta**	**XLNET**	**Bert**	**DBert**	**Albert**	**Roberta**	**XLNET**
S-DG	91.52	90.23	**92.25**	90.08	90.86	92.41	91.87	**93.78**	92.29	92.16
S-DA	**98.08**	96.44	96.72	96.57	96.15	96.03	94.45	**96.96**	94.77	95.58
S-AG	55.02	52.61	**61.86**	59.11	52.81	61.96	59.19	**66.54**	60.32	46.51
S-WA	55.07	53.61	**65.49**	48.28	53.22	49.47	47.89	**59.08**	46.53	45.49
S-BR	70.97	69.23	**78.79**	72.00	52.63	66.67	58.33	**86.67**	70.97	74.07
S-IA	84.62	86.79	84.62	**87.27**	82.61	80.70	92.00	90.57	86.79	**94.34**
S-FZ	87.18	92.68	**97.67**	90.00	84.21	92.68	95.24	92.68	**100.00**	**100.00**
T-AB	50.36	50.11	**65.88**	52.58	51.92	56.19	55.76	**70.53**	57.01	55.50
D-IA	51.16	56.60	70.37	58.18	**80.00**	76.67	80.85	80.77	78.43	**82.35**
D-DA	87.02	89.83	**95.02**	88.86	90.18	94.41	95.51	**96.32**	94.55	94.83
D-DG	83.83	84.10	**89.85**	84.48	83.92	89.87	87.53	**92.34**	91.90	91.93
D-WA	33.40	38.99	**55.37**	30.03	33.59	46.60	47.90	**61.06**	42.79	40.96

Table 3: EM-Adapter effectiveness for AutoML systems.

	AutoSklearn				AutoGluon				H2OAutoML			
	No EM-Adapter	**Attr-EM-Adapter**	**Hybrid-EM-Adapter**	**Δ**	**No EM-Adapter**	**Attr-EM-Adapter**	**Hybrid-EM-Adapter**	**Δ**	**No EM-Adapter**	**Attr-EM-Adapter**	**Hybrid-EM-Adapter**	**Δ**
S-DG	50.65	91.96	**92.92**	41.79	77.85	92.74	**93.66**	15.35	64.74	90.99	**92.50**	27.00
S-DA	92.79	**97.01**	96.32	3.87	97.62	**97.66**	96.45	-0.56	92.51	**96.79**	95.56	3.66
S-AG	44.10	59.96	**63.53**	17.65	23.28	54.39	**58.15**	32.98	36.88	56.28	**58.90**	20.71
S-WA	29.28	**58.50**	53.28	26.61	19.12	**56.72**	52.10	35.30	31.07	**55.13**	49.69	21.34
S-BR	40.00	69.28	**72.17**	30.72	0.00	67.77	**71.42**	69.60	43.24	68.72	**71.34**	26.79
S-IA	53.33	86.39	**88.28**	34.00	50.00	84.52	**90.23**	37.38	59.09	85.18	**88.88**	27.94
S-FZ	**100.00**	96.74	97.19	-3.04	71.11	97.65	**98.58**	27.01	61.90	90.35	**96.12**	31.33
T-AB	26.47	58.64	**65.90**	35.80	11.41	60.29	**64.36**	50.92	27.36	54.17	**59.00**	29.23
D-IA	64.00	59.93	**83.37**	7.65	60.87	60.69	**81.73**	10.34	62.75	63.26	**79.81**	8.79
D-DA	54.74	93.14	**95.91**	39.79	89.44	93.23	**96.15**	5.25	67.92	90.18	**95.13**	24.74
D-DG	46.79	87.11	**92.50**	43.01	69.05	88.92	**92.74**	21.78	43.01	85.23	**90.71**	44.96
D-WA	25.75	43.74	**51.15**	21.69	14.12	40.24	**49.78**	30.89	26.31	38.27	**47.86**	16.76

Table 4: Impact of EM-Adapter on AutoML performance. Bold values indicate the best configuration for a specific AutoML system; underlined values the best results for the considered dataset.

This system was then compared with the *Hybrid* variant of DeepMatcher. Table 5 reports the results of this comparison. We performed two experiments, in the first we limited the training time of the AutoML systems to 1 hour. In the second experiment, we set that time to 6 hours. The offset between the average effectiveness of AutoML systems and DeepMatcher is shown for each configuration.

Discussion. AutoML systems limited to 1 hour of training show better effectiveness than DeepMatcher in 5/12 datasets (i.e. S-BR, S-IA, T-AB , D-IA and D-WA, with an average increase of F1 equal to 9%). In the remaining cases, they generate slightly lower results, with an average F1 difference of 3.2%. However, we notice that AutoSklearn used the entire training time budget, but AutoGluon and H2OAutoML took on average a training time of 0.61 h and 0.76 h respectively. Furthermore, DeepMatcher has been trained for less than 1 hour in only 4/12 datasets. If we consider a tolerance threshold of 2%, the EM-adapted AutoML systems are comparable or outperform DeepMatcher in 9/12 datasets. This trend is further confirmed when we limit the training time to 6 hours. The average F1 score increases of 12% in the 5 datasets where the AutoML systems obtain the best results. Assuming as before a 2% tolerance threshold in F1 scores, EM-adapted AutoML systems are comparable or outperform DeepMatcher in 11/12 datasets. Also in this case, only AutoSklearn used the entire time budget for training, while AutoGluon and H2OAutoML took an average of 3.68 hours and 3.24 hours respectively (compared to 2.92 hours of DeepMatcher).

Lesson Learned. AutoML systems pipelined with *EM adapters* perform as or greater than state-of-the-art EM tools.

	DeepMatcher (Hybrid)		EM-Adapted AutoMLs (1h time budget)				EM-Adapted AutoMLs (6h time budget)			
	F1	Time (h)	AutoSklearn	AutoGluon	H2OAutoML	Δ	AutoSklearn	AutoGluon	H2OAutoML	Δ
S-DG	94.70	8.50	93.56	92.98	**93.78**	-1.26	94.02	**94.16**	93.82	-0.70
S-DA	98.40	3.75	96.64	96.75	**96.96**	-1.62	**97.08**	96.85	**97.08**	-1.40
S-AG	69.30	1.50	**68.41**	59.03	66.54	-4.64	68.41	64.53	**69.24**	-1.90
S-WA	66.90	2.17	**62.17**	58.01	59.08	-7.15	65.16	**66.12**	63.50	-1.97
S-BR	72.70	0.08	74.29	81.25	**86.67**	8.03	76.47	81.25	**86.67**	8.76
S-IA	88.00	0.25	85.71	**92.59**	90.57	1.62	91.23	92.59	**94.12**	4.65
S-FZ	100.00	0.17	**95.24**	**95.24**	92.68	-5.61	**97.67**	**97.67**	95.24	-3.14
T-AB	62.80	3.50	**76.92**	69.83	70.53	9.63	76.92	76.88	**77.06**	14.16
D-IA	74.50	0.17	**91.23**	87.27	80.77	11.92	**91.23**	**91.23**	82.35	13.77
D-DA	98.10	4.00	96.36	**97.12**	96.32	-1.50	96.36	97.12	**97.33**	-1.16
D-DG	93.80	8.50	**93.12**	92.59	92.34	-1.12	**93.53**	93.38	93.21	-0.43
D-WA	46.00	2.50	**62.80**	57.05	61.06	14.30	**67.77**	62.50	65.44	19.23

Table 5: Effectiveness of EM-Adapter with AutoML systems compared to DeepMatcher. Values in bold indicate the best configuration for a training time budget configuration.

6 CONCLUSION

The adoption of AutoML systems would make machine learning and deep learning based approaches for addressing EM tasks usable also for non-expert people. The direct application of AutoML systems to the EM problem is not possible. The reason is mainly due to the schema adopted by the datasets representing EM tasks whose records encode pairs of entities, and to the classification problem which is highly imbalanced. In this paper, we address the first problem, by introducing the *EM adapter* component which transforms the records of datasets representing entity pairs into a form which is effectively processable by AutoML systems. Our experiments show that this approach achieves a performance similar to the one of EM-task specific systems (but it does not require expert users to tune it). The future work will try to improve the performance (1) by introducing data augmentation techniques for creating more balanced training datasets for the AutoML systems; (2) by experimenting techniques for improving the embeddings (via "local embeddings" [3], generated taking into account the current dataset, and/or performing a fine-tune of the existing techniques).

ACKNOWLEDGEMENT

This work was partially funded by SBDIO I4.0 (https://www.sbdioi40.it/), an industrial research project funded by the POR FESR Emilia Romagna 2014-2020 as part of the Smart Specialization Strategy (S3).

REFERENCES

[1] Gary Anthes. 2017. Artificial intelligence poised to ride a new wave. *Commun. ACM* 60, 7 (2017), 19–21.
[2] Ursin Brunner and Kurt Stockinger. 2020. Entity Matching with Transformer Architectures - A Step Forward in Data Integration. In *EDBT*. OpenProceedings.org, 463–473.
[3] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In *SIGMOD Conference*. ACM, 1335–1349.
[4] Peter Christen. 2012. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer.
[5] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2020. An Overview of End-to-End Entity Resolution for Big Data. 53, 6, Article 127 (Dec. 2020), 42 pages. https://doi.org/10.1145/3418896
[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 4171–4186.
[7] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq R. Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed Representations of Tuples for Entity Resolution. *Proc. VLDB Endow.* 11, 11 (2018), 1454–1467.
[8] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander J. Smola. 2020. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data. *CoRR* abs/2003.06505 (2020).
[9] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. Auto-sklearn: Efficient and Robust Automated Machine Learning. In *Automated Machine Learning*. Springer, 113–134.
[10] H2O.ai. 2017. *H2O AutoML*. http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html H2O version 3.30.0.1.
[11] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-Keras: An Efficient Neural Architecture Search System. In *KDD*. ACM, 1946–1956.
[12] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. 2019. Auto-WEKA: Automatic Model Selection and Hyperparameter Optimization in WEKA. In *Automated Machine Learning*. Springer, 81–95.
[13] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep Entity Matching with Pre-Trained Language Models. *Proc. VLDB Endow.* 14, 1 (Sept. 2020), 50–60. https://doi.org/10.14778/3421424.3421431
[14] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019).
[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR (Workshop Poster)*.
[16] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD Conference*. ACM, 19–34.
[17] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet D. Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. 2019. Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey. *Artif. Intell. Rev.* 52, 1 (2019), 77–124.
[18] Matteo Paganelli, Paolo Sottovia, Francesco Guerra, and Yannis Velegrakis. 2019. TuneR: Fine Tuning of Rule-based Entity Matchers. In *CIKM*. ACM, 2945–2948.
[19] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR* abs/1910.01108 (2019).
[20] Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. 2017. Synthesizing Entity Matching Rules by Examples. *Proc. VLDB Endow.* 11, 2 (2017), 189–202.
[21] Anh Truong, Austin Walters, Jeremy Goodsitt, Keegan E. Hines, C. Bayan Bruss, and Reza Farivar. 2019. Towards Automated Machine Learning: Evaluation and Comparison of AutoML Approaches and Tools. In *ICTAI*. IEEE, 1471–1479.
[22] Jiannan Wang, Guoliang Li, Jeffrey Xu Yu, and Jianhua Feng. 2011. Entity Matching: How Similar Is Similar. *Proc. VLDB Endow.* 4, 10 (2011), 622–633.
[23] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *EMNLP (Demos)*. Association for Computational Linguistics, 38–45.
[24] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *NeurIPS*. 5754–5764.
[25] Chen Zhao and Yeye He. 2019. Auto-EM: End-to-end Fuzzy Entity-Matching using Pre-trained Deep Models and Transfer Learning. In *WWW*. ACM, 2413–2424.

Short Paper

COCOA: COrrelation COefficient-Aware Data Augmentation

Mahdi Esmailoghli
Leibniz Universität Hannover
Hannover, Germany
esmailoghli@dbs.uni-hannover.de

Jorge-Arnulfo Quiané-Ruiz
TU Berlin
Berlin, Germany
jorge.quiane@tu-berlin.de

Ziawasch Abedjan
Leibniz Universität Hannover
Hannover, Germany
abedjan@dbs.uni-hannover.de

ABSTRACT

Calculating correlation coefficients is one of the most used measures in data science. Although linear correlations are fast and easy to calculate, they lack robustness and effectiveness in the existence of non-linear associations. Rank-based coefficients such as Spearman's are more suitable. However, rank-based measures first require to sort the values and obtain the ranks, making their calculation super-linear. One of the use-cases that is affected by this is data enrichment for Machine Learning (ML) through feature extraction from large databases. Finding the most promising features from millions of candidates to increase the ML accuracy requires billions of correlation calculations. In this paper, we introduce an index structure that ensures rank-based correlation calculation in a linear time. Our solution accelerates the correlation calculation up to 500 times in the data enrichment setting.

1 INTRODUCTION

The correlation coefficient is one of the extensively used statistical measures in data science. Data scientists use the correlation coefficient to find dependencies in the data and identify possible causal relationships. In machine learning (ML) tasks, correlations can be used to evaluate the relevance of features. Correlating features expose redundancy. Thus, one can remove redundant features to avoid the curse of dimensionality. Also, a correlation between a feature and a target value can serve as a heuristic for the importance of a feature. This is exactly what filter-based feature selections are aiming for [4]: They leverage correlation to find informative and drop redundant features to achieve high accuracy in the following ML task.

Various correlation coefficients can be used to identify correlations between features in datasets. The most prominent one is the Pearson Correlation Coefficient (Pcc). Although calculating the Pcc is fairly simple and linear in the size of the compared datasets, there are several situations where a non-linear coefficient, such as the Spearman's correlation coefficient (Scc) is preferred.

Robustness. Linear correlations such as Pcc are very sensitive to outliers. As an example, Table 1 compares robustness of Pcc and Scc. Although there is no clear correlation between the two columns, the calculated Pcc identifies them as correlated, because of one outlier point (shown in red). A correlation of 1.0 is clearly misleading. Scc shows a more fitting correlation of only 0.1. This robustness makes Scc to be a better fit for noisy and dirty data such as webtables that are likely to contain outliers [7, 14].

Effectiveness. Linear correlations are not effective in capturing more complex dependencies. They are only able to find linear associations between two features. This can be useful in the case of feature selection for linear models such as linear regression. However, for complex ML models such as Random Forest or Neural Network that can train non-linear patterns, linear correlation-based feature selections can miss features with non-linear dependencies harming the achievable accuracy [23].

Table 1: Correlation in the existence of outlier.

Area (Million sq. miles)	*Calling Code*
0.29	56
0.3	90
3.8	1
0.5	51
600	9800
Pearson = 1.0	**Spearman's = 0.1**

The disadvantage of non-linear approaches, such as Scc and Kendall tau [13] is that they are rank-based and as such require a sortation of values, which poses a higher time complexity, i.e. $O(m \cdot \log(m))$ where m is the size of the variable, for calculation than their linear counterparts. The time complexity increases when we want to calculate non-linear correlation for categorical columns. The Rank-biserial correlation coefficient Rbc can be applied on one-hot encoded columns, so Rbc has a time complexity of $O(m^2)$ [7, 14]. This computation overhead negatively impacts the analysis pipeline when a large number of features have to be analyzed. For instance, to detect the redundant features, the correlation computation between each possible pair is required, which inherently leads to $O(N^2)$ correlation computations for N different features [23]. For datasets with several hundreds of potential features, the runtime overhead impedes live analysis and fast model building. For example in data enrichment [2, 17, 22, 24, 26], one aims to detect features that correlate to the given target feature from millions of extracted candidates. At this scale, runtime overhead for rank-based correlations becomes evidently a hurdle.

In this paper, we introduce a light-weight indexing structure to compute the non-linear correlation coefficient in a linear time for large-scale data enrichment tasks. It avoids the $O(m \cdot \log(m))$ sorting operation for numerical columns, benefiting the Scc calculation, and avoiding the $O(m^2)$ complexity of dealing with the one-hot encoding of categorical columns for Rbc. Our light-weight index also enables Cocoa (our system) to scale to the massive number of external tables. Furthermore, the nature of the correlation calculation also enables Cocoa to perform light-weight joins instead of full materialization of joins with candidate columns. In summary, we make three major contributions:

(1) We introduce a new index structure that enables us to compute the non-linear correlation coefficient in linear time and generalizes also for enrichment through partial joins where ranks are missing and have to be adapted.
(2) We propose algorithms that leverage our index structure to detect correlations between numerical and categorical columns to a target column in linear time.
(3) We introduce a correlation-based data enrichment solution that increases the accuracy of the ML model for a user-defined task compared to other enrichment solutions.

2 PROBLEM STATEMENT

Given an input dataset D with m rows, two explicitly selected columns q and t from D, so-called query and target columns respectively, and a corpus of external tables $T = \{T_1, ..., T_n\}$, the goal is to enrich D with the top-k_c columns from any table $T_i \in T$ that correlate with t. As we focus on regression tasks, t is a numerical column while features can be either numerical or categorical. The query column q is used to find the related tables, i.e., tables that are joinable with D on q. Ideally, q is an identity exposing column, such as name or ID. Without loss of generality, we thus assume q is selected explicitly by the user.

Depending on whether an extracted feature is numerical or categorical, one has to use Scc or Rbc, respectively. Equation 1 shows the formula of Scc where x_i and y_i are the i^{th} values in the first and second column respectively. Both columns are of size m. $R(x_i)$ represents the rank of value x_i. For instance, $R(8) = 2$ in list $[8, 11, 4, 9]$ because the value 8 is second in the sorted list $[4, 8, 9, 11]$. $\overline{R(x)}$ and $\overline{R(y)}$ represent the average of ranks in the first and the second column respectively.

$$\text{Scc} = \frac{\sum_{i=1}^{m}(R(x_i) - \overline{R(x)})(R(y_i) - \overline{R(y)})}{\sqrt{\sum_{i=1}^{m}(R(x_i) - \overline{R(x)})^2 (R(y_i) - \overline{R(y)})^2}} \quad (1)$$

The Scc calculation for numerical columns is in $O(m \cdot \log m)$ because one has to obtain the ranks through sortation.

To compute the correlation between categorical columns and t, the Rbc calculates the Scc between each one-hot feature of the categorical column and t [3, 9, 18]:

$$\text{Rbc} = \frac{M_1 - M_0}{std} \sqrt{\frac{n_1 n_0}{m^2}} \quad (2)$$

M_1 and M_0 represent the average target rank for 1s and 0s in a one-hot feature, n_0, n_1 are the number of ones and zeros in the one-hot feature, and std and m define the standard deviation of target ranks and the number of values in columns, respectively. In our case, m is the number of rows in the input dataset. The overall correlation is then the maximum Rbc between each one-hot feature and t Based on Equation 2, we can calculate the Rbc in $O(m)$ time complexity per one-hot feature. As a column can have up to m unique values, thus the overall complexity is $O(m^2)$.

If the join operation is implemented as a hash-based join, the join task has a time complexity of $O(m)$ per table. The follow-up correlation calculation per column has either a complexity of $O(m \cdot \log m)$ or $O(m^2)$ depending on whether the feature is numerical or categorical. In retrieval tasks with large databases, we can observe the runtime is dominated by the correlation calculation.

Problem We are thus looking for an index structure that allows us to calculate both Scc and Rbc in linear time. To be able to index data repositories in the scale of web tables, our index structure should be simple and light-weight.

3 COCOA SYSTEM

Figure 1 depicts the abstract view of the components designed in our system Cocoa. The main components are *Table Finder*, *Join Mapper*, and *Data Augmenter*. A user provides Cocoa with a dataset D and specifies its query column q and ML target column t. At last, it returns the top-k_c most correlating columns as the output of the system. Now, we describe each of these components.

Table Finder. This component uses q and the inverted index of the DataXformer system [1] to obtain top-k_t joinable tables.

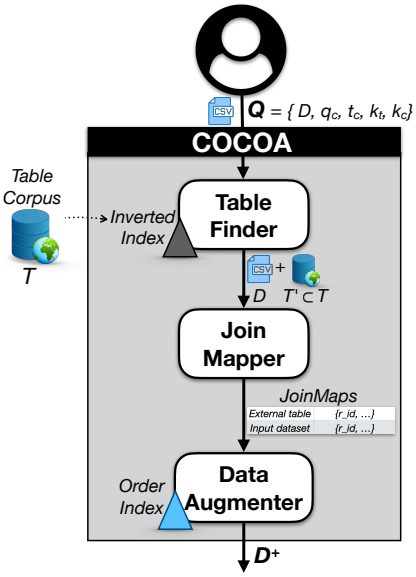


Figure 1: The overall architecture of Cocoa

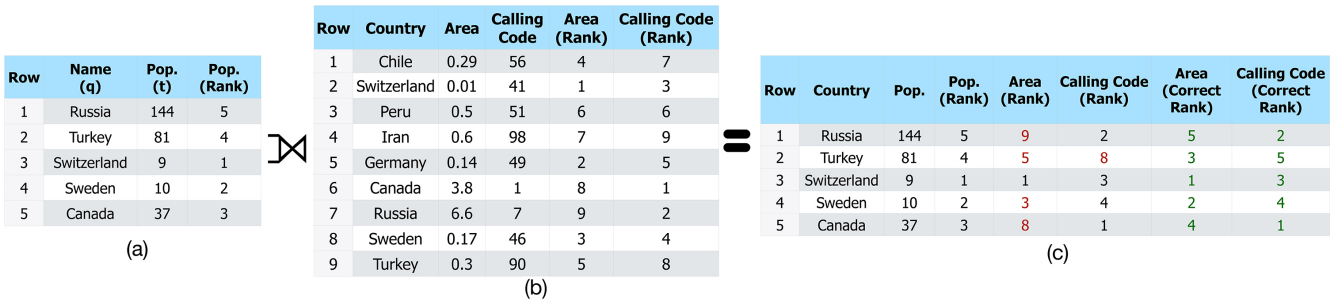
A table is joinable if it contains at least one column that overlaps values with q. For each value v_i in external tables, the inverted index lists the coordinates of its containing tables: $v_i \mapsto \{(T_{i1}, C_{i1}, P_{i1}), (T_{i2}, C_{i2}, P_{i2}), ...\}$, where T_{ij}, C_{ij}, and P_{ij} are identifiers of tables, columns, and rows in the table corpus, respectively. We store the inverted indices inside a column store and generate a SQL query to find joinable external tables in parallel and using database-level optimizations. This approach allows to push the process of finding the joinable tables down to the database itself. The SQL query calculates the overlap between each external column and q and then selects the top-k_t tables by sorting them based on the overlap score.

Running example Figure 2(a) depicts a user-provided dataset D for the task of predicting the population of a country. The column with the country names will serve as the query column q and the population as the ML target column t. Figure 2(b) depicts an external table (T_1) that contains three columns: *Country*, *Area*, and *Calling Code*. In our example, the index entry for the values "Germany" in *Country* would be: $Germany \mapsto \{(T_1, Country, 5)\}$.

Join Mapper. The *Join Mapper* receives a set of joinable tables T' as input and *virtually* joins them with the input dataset D. To enrich the input dataset with external tables, we have to apply a LEFT JOIN because we need to keep information in D and add only overlapping information from the external tables [6]. Instead of a complete materialized LEFT JOIN, our *Join Mapper* generates a lightweight JoinMap, which is inspired by P.Valduriez's "Join Indices" [19]. Using our *Order index*, we can use the light-weight join to also calculate the correlations before materializing the join. Thus we limit the join materialization for relevant columns with high correlation and further speed up the process.

Figure 3 depicts the JoinMap for the example in Figure 2. *Country* column in T_1 is overlapping with q in D. The corresponding map shows which row in *Country* has the same respective value in q. As it is shown, the value "Switzerland" in row 2 of *Country* appears in row 3 of q. In the case of duplicates in q, respective cells in the JoinMap will contain more than one value.

Data Augmenter. This component receives the JoinMaps as input and uses a novel structure, called *Order index*, to efficiently compute the Scc and Rbc. It evaluates the external columns based on their correlation with t and enriches the input dataset D with the top-k_c correlating columns to generate the final dataset D^+.



Row	Name (q)	Pop. (t)	Pop. (Rank)
1	Russia	144	5
2	Turkey	81	4
3	Switzerland	9	1
4	Sweden	10	2
5	Canada	37	3

(a)

Row	Country	Area	Calling Code	Area (Rank)	Calling Code (Rank)
1	Chile	0.29	56	4	7
2	Switzerland	0.01	41	1	3
3	Peru	0.5	51	6	6
4	Iran	0.6	98	7	9
5	Germany	0.14	49	2	5
6	Canada	3.8	1	8	1
7	Russia	6.6	7	9	2
8	Sweden	0.17	46	3	4
9	Turkey	0.3	90	5	8

(b)

Row	Country	Pop.	Pop. (Rank)	Area (Rank)	Calling Code (Rank)	Area (Correct Rank)	Calling Code (Correct Rank)
1	Russia	144	5	9	2	5	2
2	Turkey	81	4	5	8	3	5
3	Switzerland	9	1	1	3	1	3
4	Sweden	10	2	3	4	2	4
5	Canada	37	3	8	1	4	1

(c)

Figure 2: Running example: (a) Input dataset D; (b) External table T_1; (c) Joined result of D and T_1

row in C_1	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
row in q	∅	3	∅	∅	∅	5	1	4	2

Figure 3: An example of JoinMap

4 CORRELATION CALCULATION

The main goal of the *Data Augmenter* is to calculate the Scc and Rbc between each provided external column and the target column t in $O(m)$. Thus, we want to refrain from sorting the values of every potential numerical column and the generation of one-hot encodings for every categorical column during the extraction time. One might think of a simple solution that calculates the ranks and one-hot encodings offline and maintains them as part of the inverted index. However, maintaining the ranks will lead to calculation errors of the Scc when related tables are only partially joinable. Columns *Area (Rank)* and *Calling Code (Rank)* in Figure 2(c) show this error in obtaining the ranks after the partial join. Non-consecutive ranks lead to incorrect Scc. For instance, the correct Scc between *Calling Code* and t is -0.1, but using the incomplete ranks returns the Scc of $+0.117$. Furthermore, storing the one-hot encoding of all columns would unnecessarily blow up the index size. Therefore, we propose algorithms based on a new *Order index* that keeps track of the order of each value in the numerical columns and efficiently generates one-hot encodings for categorical columns.

4.1 Order Index Structure

We now introduce the simple, yet effective, *Order index* structure. Instead of the actual ranks, it stores the relative order of column values enabling the system to compute Scc and Rbc in a linear time. Keeping track of the order of values enables us to calculate the correct ranks on-demand despite partial joins. Also, keeping relative orders enables us to walk through all non-zero values in one-hot features in the right order to calculate the Rbc for categorical columns. To maintain the order information in a concise way, the *Order index* maps each pair of a table identifier T_{id} and a column identifier C_{id} to the column values as follows:

$$T_{id}, C_{id} \mapsto \{s,\ (o_1,\ ...,\ o_r),\ (b_1,\ ...,\ b_r)\} \tag{3}$$

$$s \leftarrow \alpha, \quad where \quad R(v_\alpha) = 1 \tag{4}$$

$$b_i = \begin{cases} False, & R(v_i) = R(v_{o_i}) \\ True, & R(v_i) \neq R(v_{o_i}) \end{cases} \tag{5}$$

Each entry in the *Order index* consists of the starting point s, which represents the row id of the minimum value in the column (α in Equation 4). v_i is the value located in the i^{th} row. The rank of the minimum value is always 1. In categorical columns, the minimum is the first value in the alphabetically sorted list. The second item (order list), contains a list of values where o_i is the row id of the next greater value than v_i. In case of having repetitive values, o_i is the row id of the next equal value. If v_i is the last value in the sorted list, $o_i = -1$. The item b_i denotes whether v_{o_i} is greater than its predecessor v_i or not (Equation 5). We use the same index for both numerical and categorical columns. To distinguish the numerical and categorical columns during the correlation calculation, we use a simple heuristic: a column is considered as categorical if it contains at least one non-numeric value. We store it as an additional bit per column. The *Order index* of the column *Area* in our running example in Figure 2(b) would be:

$\{2,\ (9,\ 5,\ 4,\ 6,\ 8,\ 7,\ -1,\ 1,\ 3), (T,\ T,\ T,\ T,\ T,\ T,\ \emptyset,\ T,\ T)\}$

In column *Area*, the minimum is located in row 2 (0.01), so, $s = 2$. The second element of the index represents a list of pointers to the next greater (or equal) values. The first item in this list o_1 points to row id 9 and it means that the next greater value "0.3" after the first value "0.29" is stored in the 9^{th} row. Likewise, the 9^{th} pointer o_9 in the order list is 3, which means that the next greater value "0.5" is located in row 3. Notice that as there are no repetitive values in column *Area*, all of the binary values in the third index element, except for the maximum value pointer, are *True*.

Figure 4 shows the visualized representation of the *Order index* for the column *Area* in T_1. Each square represents one value in the external column. The blue square represents the minimum value and the red one shows the maximum. Each edge that connects one square to another depicts the relative order o_i. Small numbers in the circles are the row ids of the values in the column *Area*. Each value has an outgoing edge except the maximum value. Labels on each edge represent the index values stored for the source value. For example, the outgoing edge from the 4^{th} item in the list, is $[6, T]$, which translates to $o_4 = 6$ and $b_4 = T$.

The *Order index* is the most basic index structure that supports correlation calculation in $O(m)$. A more complex B+ tree adaptation can be used if frequent corpus updates are expected. Here, we exclude the tree benefits for a more space-efficient index.

4.2 SCC for Numerical Columns

We use an example to describe how we leverage the underlying *Order index* to calculate the Scc between numerical columns of an external table and t. Figure 4 shows the *Order index* of the *Area* column and the JoinMap of the table T_1 from our running example. Assume that `p` is a pointer and in the beginning, `p` references the minimum value in the column. In each iteration, `p` traverses through the available links until it reaches the red square which means that the ranking process is finished.

Starting from the minimum value "0.01", the algorithm checks the `JoinMap` for any mapping from the value in *Area* to a value in q. There is a mapping from "0.01" to the value in the 3^{rd} row of

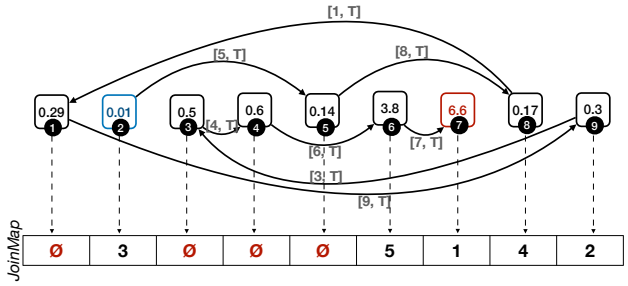


Figure 4: Visual representation of the Order index for the column *Area* from the running example in Figure 2.

q. Therefore, initial rank, which is 1, is assigned to the 3^{rd} value of *Area* (see *Area (Correct Rank)* in Figure 2(c)). Then, p will be updated to the next greater value. Here, "0.01" is linked to the 5^{th} value, which is "0.14". However, there is no map from "0.14" to any value in q. Thus, p will be updated based on the outgoing link from "0.14" to the next greater value located in the 8^{th} square with the value of "0.17". There is a mapping from the value "0.17" to the row with id 4, so the next rank, which currently is 2, will be assigned to the 4^{th} value in *Area*. This process continues until the pointer p reaches the 7^{th} and the maximum value. Since there are no outgoing links, the iteration is finished. The ranks for both *Area* and *Calling Code* columns are shown in the Figure 2(c) as *Area (Correct Rank)* and *Calling Code (Correct Rank)*, respectively.

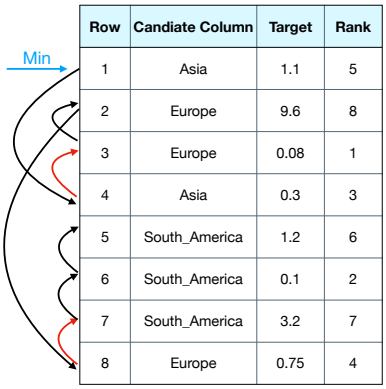
According to Equation 1, we can indeed compute the Scc in $O(m)$ by having correct ranks. Obtaining the ranks is now also possible in $O(m)$. The algorithm iterates over the values in the ordered list only once. Therefore, if the time complexity of the operations per value is constant and the path from the minimum to the maximum value is cycle-free, we conclude that the Scc calculation is done in $O(m)$. First, all operations of reading the *JoinMap*, assigning/updating current rank, and updating p to the next value are done in a constant time because the *JoinMap* and the *Order index* items are implemented using array structure and accessing the array cells is performed in a constant time using the available cell index. We also know that each o_i value, which is represented by an edge, refers to a unique row id even in the existence of repetitive values. Thus, the path from the minimum to the maximum value is cycle-free. The path length is r, i.e., the average number of rows in the external columns. External tables are often smaller than the input datasets ($r << m$), e.g., the average external table size is 10 for DWTC and 1540 for open data [26]. Thus, the complexity is typically bound by m.

4.3 RBC for Categorical Columns

To calculate the dependency between external categorical columns and t, we have to generate the one-hot features of the column and calculate the Rbc between each generated feature and the target rank column. Using our index we can avoid this and simulate the calculation in linear time. The *Order index* allows us to walk through the non-zero values in the one-hot features, one feature at a time. By iterating non-zero values of each one-hot feature consecutively we can compute the Rbc for all possible feature in one pass of reading the categorical column using the following derived Equation 6, which is obtained by replacing M_0 and n_0 in Equation 2 with $\frac{S-M_1 n_1}{n_0}$ and $m - m_1$ respectively:

$$\text{Rbc} = \frac{m \cdot s_1 - n_1 \cdot S}{std \cdot m \cdot \sqrt{n_1(m - n_1)}} \tag{6}$$

In Equation 6, S is the sum of all the ranks in the target column and s_1 the sum of the ranks where the corresponding one-hot value is one. Using Equation 6, variables m, S, and std are calculated once. To compute the Rbc between each one-hot feature and t, it is enough to compute s_1 and n_1 for each one-hot feature.



Row	Candiate Column	Target	Rank
1	Asia	1.1	5
2	Europe	9.6	8
3	Europe	0.08	1
4	Asia	0.3	3
5	South_America	1.2	6
6	South_America	0.1	2
7	South_America	3.2	7
8	Europe	0.75	4

Figure 5: Categorical candidate column example.

Cocoa uses the *Order index* to iterate over the sorted list of categorical values. It is able to compute s_1 and n_1 per one-hot feature without generating the features because the *Order index* allows to read the repetitions of the values consecutively. Once the iteration reaches a different value, b_i announces the end of the current one-hot feature. At this point, the Rbc is calculated based on Equation 6. In the end, the maximum Rbc will be reported.

Figure 5 shows an example for Rbc calculation. It depicts a categorical column "Candidate Column" and the target column "Target". We would like to compute the Rbc of all one-hot features in one pass of reading the values. The ranks of the target values are shown in the "Rank" column. The location of the minimum value is represented in blue and each link depicts the location of the next value. The red-colored links show that the next value is greater, i.e., different, than the current value. Obtaining the standard deviation of the "Rank" column and m, we only need to calculate s_1 and n_1 for each unique value.

Starting with the minimum value, as long as the followed arrow is not red, we continue reading rank values and adding them to s_1, increasing n_1 by 1 in each iteration. After reading the value in row number 1, s_1 and n_1 are equal to 5 and 1 respectively. Following the links gets us to the 4^{th} row. Reading the rank value, s_1 is increased to 5+3 = 8 and n_1 to 1+1 = 2. The next arrow is red and it means that the following value is different from the current one. Therefore, we calculate the correlation using Equation 6. Repeating this process, another correlation is calculated in row number 7. Reaching the last row, we compute the final correlation and report the maximum Rbc. Rbc calculation is done in $O(m)$ as the algorithm iterates over the ordered list only once.

5 EXPERIMENTS

We carried out a series of experiments to evaluate Cocoa with the following questions in mind: (i) What is the performance gain through *Order index* for calculating feature correlations on large corpora? (ii) How efficient is the light-weight virtual join? (iii) How scalable is Cocoa? Before we delve into the detail, we first describe the setup of our experimental evaluation.

5.1 Data and Experimental Setup

We tested our approach on top of several open databases. The Dresden webtable corpus (DWTC)[1] contains more than 145M tables and 870M unique columns. The Canada, US, and Uk Open Data corpus used in prior work [26] contains more than 745,000 columns and 14,000 tables. It is noteworthy that the Open Data

[1] https://wwwdb.inf.tu-dresden.de/misc/dwtc/

Table 2: Experimental datasets.

Dataset	*Query Column*	*Target*	*Instances*
Kaggle Open Food Facts	Product Name	Energy per 100g	356,000
Kaggle NYC Airbnb	Place Name	Price	48,000
Cities	City Name	Population	40,000
Kaggle Video Game Sales	Game	Global Sales	16,599
PageView	Name	Visit	11,000
Kaggle IMDB	Movie Title	IMDB Score	5,043
Presidential	County	Votes	3,000
Kaggle Craft Beers	Name	Alcoholic content	2,400
Kaggle Human Freedom Index	Country	Homicide Index	1,450
Kaggle World Happiness Report	Country/Region	Score	157
University	Name	World Ranking	100

corpus only contains numerical data. We ran our experiments on a server with 28 CPU cores and 250GBs of main memory. We implemented our solution in *Python 3.7* and used Vertica *v9.1.1-0* [16] as the storage. Implementations and datasets are available in our GitHub repository[2]. All compared approaches use the same table retrieval module. They only differ with regard to the join, filtering, and correlation calculation steps.

Table 2 shows the datasets that we chose from different domains as query datasets. For the Open Data corpus, we use the benchmark query columns provided in the *Josie* paper [26].

Baselines. For these experiments, we compare Cocoa with three baselines and two combined versions of Cocoa:

(1) Sort-based enrichment (Sbe). Sbe calculates the Scc and Rbc without using the provided order index. It has to sort online and create one-hot encoding of categorical columns. The comparison with this baseline allows us to evaluate the efficiency of the *Order index* in the enrichment pipeline.

(2) TR. It is a rule-based method that skips joins with non-informative tables before any further calculation [15].

(3) TR+Cocoa. TR is a complementary approach to Cocoa. TR drops the non-informative tables before joining. Then, Cocoa extracts the correlating columns from the tables.

(4) RF. Here, we apply the random forest feature selection (RF) on top of the overlap-based enrichment method [26]. In this method, we join k_t related tables and RF picks the most informative features with respect to the ML task.

(5) Cocoa+RF. Here, Cocoa enriches the input data with the top 100 correlated columns and delivers to RF.

5.2 Results

We start with the DWTC experiments. Figure 6(a) shows the average runtime of Cocoa, Sbe, and TR applied on all query datasets. The average runtime is shown for varied k_t values. In this experiment, we drop the break-down based on k_c because it has only a negligible improvement on the runtime of Cocoa and no impact on the other approaches. The depicted runtime includes the time for joining tables and calculating the Scc/Rbc. In the case of TR, runtime represents joining and rule validation time. Note that the runtime is depicted in logarithmic scale.

Cocoa outperforms Sbe on all datasets and for all different k_t values and ultimately, we see better performance on average. Cocoa can be up to 520X faster than Sbe using the introduced *Order index*. Cocoa, on average, is slower than *TR*. *TR* applies a rule-based table filter before joining the tables. It computes the cardinality of the tables and decides whether the join is safe to skip or not. Rules in TR - unlike Cocoa- are applied per external table and not per columns. Therefore, the rule verification in TR is computationally less expensive. The coarse-granular filtering comes at the cost of effectiveness [6]. According to Figure 6(a), the runtime of all approaches increases with k_t because more external tables have to be processed for the Scc calculation.

[2]https://github.com/BigDaMa/COCOA

Figure 6(b) shows the runtime experiment on Open Data. Here, k_t has a lower impact compared to the DWTC corpus, because of the small number of tables. So, we consider all tables that have overlap with q as candidate join tables. As expected, similar to the DWTC, Cocoa is faster than Sbe due to the fast correlation calculation through our introduced index structure. However, TR is surprisingly slower than the other two systems. The reason is that in comparison to DWTC, the tables in the Open Data benchmark, are not sparse and yield higher overlap between queries and the tables. Therefore, the TR rule is passed and most of the tables are joint. In this case not only does TR not provide any additional runtime benefits but also it introduces cardinality calculation overhead with almost zero pruned external table.

Figure 6(c) compares Cocoa to three hybrid system on DTWC with $k_t = 1000$. We build these hybrid systems to evaluate the impact of the feature pre-filtering on the runtime of the systems and determine the fastest system combination that considers feature-target association for the ML task. The combination *TR*+Cocoa results in better performance than every other strategy. TR drops the non-informative tables and reduces the search space for Cocoa to find the correlating features in a faster way. This experiment shows that although *TR* does not perform well with regard to effectiveness [6], it can be a complementary heuristic for Cocoa to reduce the search space and enrich the input dataset much faster than any other solutions at hand.

RF is the slowest approach on all datasets except the *University* dataset. This is due to the tremendous number of features that are delivered to *RF*. Note that experiments that are not finished within 8 hours are underestimated with 8 hours in the runtime. RF fails to terminate for 4 datasets: *Cities*, *IMDB*, *Food*, and *NYC Airbnb*. Pruning the search space using Cocoa improves runtime for RF. Cocoa +RF only fails to finish on the *Food* dataset.

We notice that *TR*+Cocoa performs even better than TR on larger datasets, such as *Food*. The reason is that TR materializes all joins but Cocoa uses the light-weight virtual joins that are much faster because only the top-k_c columns are materialized.

Scalability. To better evaluate the scalability of Sbe- and Cocoa-based solutions with regard to the dataset size, we measure the average runtime for a single correlation calculation in both approaches. Figure 7 shows this experiment on the *City* dataset scaling the number of rows. The average calculation time increases much faster for Sbe than for Cocoa. On a dataset with $1M$ rows, the runtime difference between the two approaches for one correlation calculation is about 118 seconds. This difference is crucial in the scale of thousands of external tables. In our experiments, for the *City* dataset the number of correlation calculations easily exceeds 52, 000 calculations to enrich the dataset, thus this scalability issue will lead to serious runtime problems. In this case, Sbe would need 71 days to complete the task while Cocoa would require only 9.5 minutes. Notably, the variances in the graph are negligible and at most 0.0001 milliseconds.

Index Size. We only store the row ids and bits instead of actual values, therefore, indexing the DWTC corpus requires less than 12GB disk storage compared to almost 300GB database size.

6 RELATED WORK

Data enrichment It is referred to the line of research that expands an input dataset using external sources such as webtables [11, 22, 24], Open data [17, 26], or knowledge bases [5].

Most pieces of work from the database community focus on finding joinable tables. For this purpose, some heuristics, such as

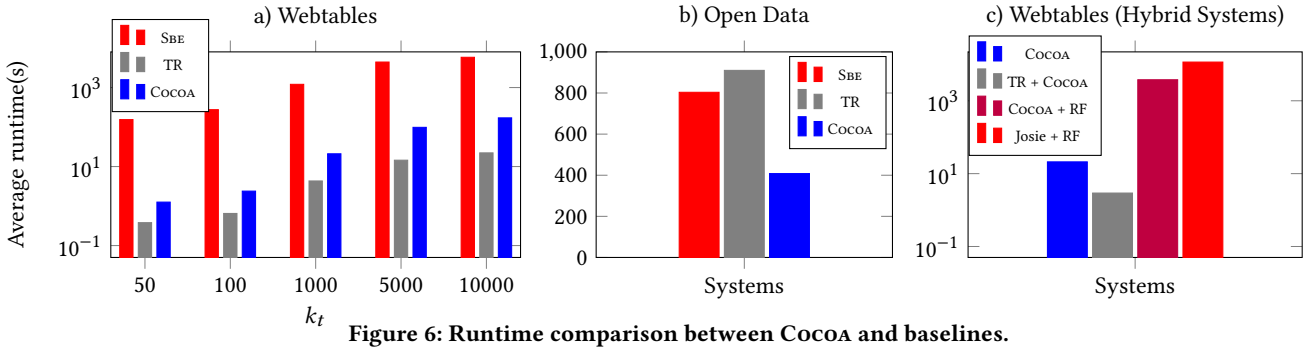


Figure 6: Runtime comparison between COCOA and baselines.

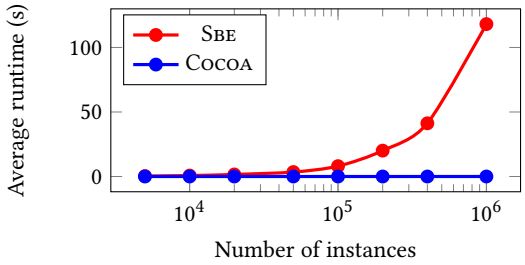


Figure 7: Scaling SCC and RBC on the *City* dataset.

overlap similarity [21, 25, 26] or a combination of the coverage and value consistency [8, 11] is used to find candidate tables for enrichment. These works do not consider the downstream ML task for the retrieval process and only rely on the joinability of the extracted tables. Furthermore, they assume that the external tables can fit in the main memory. Kumar et al. address the problem of data enrichment by defining a set of decision rules to prevent the joins that will not contribute to higher accuracy in the downstream ML models [15]. Their approach skips the joins to achieve the minimum information loss. Therefore, the dropping rate is quite low and in the scale of a large corpus of tables, the final enriched dataset will still contain a large number of columns, that have to be handled in the time-consuming feature selection process [23]. In COCOA, we propose an ML-aware data enrichment solution. It leverages the SCC/RBC to find the most promising columns for the downstream ML task. ARDA [6] is an enrichment system that leverages sampling techniques to find the informative joins and then uses an ensemble feature selection method to select the best features. Its feature selection algorithm RF focuses on accuracy and is not efficient when it comes to a large number of external tables.

Feature Selection Feature selection algorithms [6, 10, 12, 20] are designed to find the best feature set for a specific ML task after the enrichment process. In this paper, we aim to blend the feature selection with the extraction phase that can speed up the traditional *Extract-Then-Select* pipeline. We also discussed a combination of COCOA with the most promising feature selection method [6] in our experiments.

7 CONCLUSION

We presented COCOA, a new data enrichment system. It enables the efficient calculation of non-linear correlation coefficients to select the most correlating features for a user-defined ML task. In particular, we introduced an index structure that allows to calculate non-linear correlation coefficients in linear time complexity. COCOA is designed to be general and hence it can be complemented with other table-based filters or used for any analytic task that depends on value rankings and rank-based scores.

Acknowledgements. This project has been supported by the German Research Foundation (DFG) under grant agreement 387872445 and the German Ministry for Education and Research as BIFOLD — "Berlin Institute for the Foundations of Learning and Data" (01IS18025A and 01IS18037A).

REFERENCES

[1] Ziawasch Abedjan, John Morcos, Michael N Gubanov, Ihab F Ilyas, Michael Stonebraker, Paolo Papotti, and Mourad Ouzzani. 2015. Dataxformer: Leveraging the Web for Semantic Transformations.. In *CIDR*.
[2] Bortik Bandyopadhyay, Xiang Deng, Goonmeet Bajaj, Huan Sun, and Srinivasan Parthasarathy. 2019. Automatic Table completion using Knowledge Base. *arXiv preprint arXiv:1909.09565* (2019).
[3] Douglas G Bonett. 2019. Point-biserial correlation: Interval estimation, hypothesis testing, meta-analysis, and sample size determination. *Brit. J. Math. Statist. Psych.* (2019).
[4] Girish Chandrashekar and Ferat Sahin. 2014. A survey on feature selection methods. *Computers & Electrical Engineering* 40, 1 (2014), 16–28.
[5] Weiwei Cheng, Gjergji Kasneci, Thore Graepel, David Stern, and Ralf Herbrich. 2011. Automated feature generation from structured knowledge. In *CIKM*. 1395–1404.
[6] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. 2020. ARDA: Automatic Relational Data Augmentation for Machine Learning. *PVLDB* 13, 9 (2020).
[7] Christophe Croux and Catherine Dehon. 2010. Influence functions of the Spearman and Kendall correlation measures. *Stat. meth. & app.* 19, 4 (2010), 497–515.
[8] Anish Das Sarma, Lujun Fang, Nitin Gupta, Alon Halevy, Hongrae Lee, Fei Wu, Reynold Xin, and Cong Yu. 2012. Finding related tables. In *SIGMOD*. 817–828.
[9] Hakan Demirtas and Donald Hedeker. 2016. Computing the point-biserial correlation under any underlying continuous distribution. *Communications in Statistics-Simulation and Computation* 45, 8 (2016), 2744–2751.
[10] Włodzisław Duch, Tomasz Winiarski, Jacek Biesiada, and Adam Kachel. 2003. Feature selection and ranking filters. In *ICANN/ICONIP*, Vol. 251. 254.
[11] Julian Eberius, Maik Thiele, Katrin Braunschweig, and Wolfgang Lehner. 2015. Top-k entity augmentation using consistent set covering. In *SSDBM*. 8.
[12] Isabelle Guyon and André Elisseeff. 2003. An introduction to variable and feature selection. *JMLR* 3, Mar (2003), 1157–1182.
[13] Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.
[14] Yunmi Kim, Tae-Hwan Kim, and Tolga Ergün. 2015. The instability of the Pearson correlation coefficient in the presence of coincidental outliers. *Finance Research Letters* 13 (2015), 243–257.
[15] Arun Kumar, Jeffrey Naughton, Jignesh M. Patel, and Xiaojin Zhu. 2016. To join or not to join?: Thinking twice about joins before feature selection. In *SIGMOD*. 19–34.
[16] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *PVLDB* 5, 12 (2012), 1790–1801.
[17] Fatemeh Nargesian, Erkang Zhu, Ken Q Pu, and Renée J Miller. 2018. Table union search on open data. *PVLDB* 11, 7 (2018), 813–825.
[18] Robert F Tate. 1954. Correlation between a discrete and a continuous variable. Point-biserial correlation. *The Annals of mathematical statistics* 25, 3 (1954), 603–607.
[19] Patrick Valduriez. 1987. Join indices. *TODS* 12, 2 (1987), 218–246.
[20] Jason Weston, Sayan Mukherjee, Olivier Chapelle, Massimiliano Pontil, Tomaso Poggio, and Vladimir Vapnik. 2001. Feature selection for SVMs. In *NeurIPS*. 668–674.
[21] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. 2009. Top-k set similarity joins. In *ICDE*. IEEE, 916–927.
[22] Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. 2012. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD*. 97–108.
[23] Lei Yu and Huan Liu. 2003. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *ICML-03*. 856–863.
[24] Meihui Zhang and Kaushik Chakrabarti. 2013. InfoGather+ semantic matching and annotation of numeric and time-varying attributes in web tables. In *SIGMOD*. 145–156.
[25] Yi Zhang and Zachary G Ives. 2020. Finding Related Tables in Data Lakes for Interactive Data Science. In *SIGMOD*. 1951–1966.
[26] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD*. 847–864.

Short Paper

Efficient Exploratory Clustering Analyses with Qualitative Approximations

Manuel Fritz, Dennis Tschechlov, Holger Schwarz
University of Stuttgart
Stuttgart, Germany
{manuel.fritz,dennis.tschechlov,holger.schwarz}@ipvs.uni-stuttgart.de

ABSTRACT

Clustering is a fundamental primitive for exploratory data analyses. Yet, finding valuable clustering results for previously unseen datasets is a pivotal challenge. Analysts as well as automated exploration methods often perform an exploratory clustering analysis, i.e., they repeatedly execute a clustering algorithm with varying parameters until valuable results can be found. k-center clustering algorithms, such as k-Means, are commonly used in such exploratory processes. However, in the worst case, each single execution of k-Means requires a super-polynomial runtime, making the overall exploratory process on voluminous datasets infeasible in a reasonable time frame. We propose a novel and efficient approach for approximating results of k-center clustering algorithms, thus supporting analysts in an ad-hoc exploratory process for valuable clustering results. Our evaluation on an Apache Spark cluster unveils that our approach significantly outperforms the regular execution of a k-center clustering algorithm by several orders of magnitude in runtime with a predefinable qualitative demand. Hence, our approach is a strong fit for clustering voluminous datasets in exploratory settings.

1 INTRODUCTION

Clustering is a fundamental primitive for exploratory tasks. Jain identified three main general purposes of clustering, which emphasize the exploratory power of clustering analyses [15]: (1) Assessing the structure of the data. Here, the goal is to exploit clustering to gain a better understanding of data, to generate hypotheses or to detect anomalies. (2) Grouping entities. Clustering aims to group similar entities into the same cluster. Thus, previously unseen entities can be assigned to a specific cluster. (3) Compressing data, i.e., to use clusters and their information as summary for further steps.

Due to their runtime behavior, k-center clustering algorithms, such as k-Means [16], k-Medians [6] or k-Mode [14] are commonly used [18], especially on voluminous data. However, the expected number of clusters k has to be provided prior to their execution. Particularly for previously unknown datasets, choosing this parameter is a tremendous pitfall and requires particular caution: Wrong values lead to bad results regarding the above-mentioned purposes, i.e., imprecise structurings, groupings or compressions are performed, thus making the clustering results unusable in the worst case.

In order to achieve valuable clustering results, k-center clustering algorithms are typically executed with varying values for k. This can be performed manually by analysts or in an automated manner by exploration methods, which perform an automated exploratory process in order to find valuable clustering results [9]. However, as these approaches require several complete runs of a clustering algorithm, they tend to be very time-consuming, in particular when the clustering algorithm is repeatedly executed on voluminous datasets [11].

Regarding k-Means as instantiation of a k-center clustering algorithm, the runtime for a single execution is $O(knd\omega)$ [13], where k is the number of clusters, n is the number of entities in a dataset, d is the number of dimensions, and ω is the number of clustering iterations performed. In the worst case, i.e., when performing k-Means until convergence, ω is super-polynomial regarding n [3], i.e., a single execution of the clustering algorithm on large datasets is already very time-consuming. However, for exploratory clustering analyses, where clustering results of several parameter values are of interest, analysts typically require fast, yet adequately accurate approaches that can be used to gain a fundamental understanding of the results. As we show in this paper, efficient exploratory clustering analyses are possible by controlling the number of clustering iterations in the exploration process for promising parameter values.

Many implementations, such as sklearn[1] or Spark's MLlib[2], allow to reduce the runtime of clustering algorithms by setting a fixed threshold for the number of clustering iterations. However, it is not clear how to set this threshold such that a valuable clustering result can be achieved, since this choice highly depends on dataset characteristics and the initialization of a clustering algorithm. Hence, too few clustering iterations lead to an imprecise clustering result, whereas too many clustering iterations lead to an unacceptable runtime. In this work, we introduce a novel approach to efficiently terminate k-center clustering algorithms as soon as a predefined qualitative demand is met, thus reducing the number of clustering iterations and therefore also the runtime.

Our contributions include the following:

- We propose our novel approach to terminate k-center clustering algorithms based on a predefined qualitative demand, which is typically defined by analysts.
- We show, that our approach (i) provides negligible runtime overhead for its calculations, and (ii) can be seamlessly integrated into exploratory clustering analyses.
- The results of our comprehensive evaluation on a distributed Spark cluster unveil that our approach outperforms a regular execution of a k-center clustering algorithm with speedups of several orders of magnitude, while the given qualitative demand is met in most cases.

The remainder of this paper is structured as follows: We present related work in Section 2. In Section 3, we analyze advantages and pitfalls of a closely related approach to reduce the number of clustering iterations. Subsequently, we present our novel generic qualitative approximation approach for k-center clustering algorithms in Section 4. In Section 5, we summarize the results of a comprehensive evaluation unveiling the benefits of our method. Finally, we conclude this work in Section 6.

[1] https://git.io/Jt8lJ [2] https://git.io/Jt8lt

2 RELATED WORK

All k-center clustering algorithms proceed in an iterative manner, i.e., the same sequence of steps is repeated until a given convergence criterion is met. Each clustering iteration comprises the following three steps: (1) initialize or change the position of the centroids, which are centers of gravity for a specific cluster, (2) improve the clustering by (re-)assigning entities to the closest centroid, and (3) check for convergence. Eventually, many clustering algorithms terminate when no entities change their membership anymore. As mentioned above, k-Means as concrete instantiation of such an algorithm has the runtime complexity of $O(knd\omega)$ [13]. Since clustering results for several values for k are at the core of exploratory processes, we focus on related work, which addresses the remaining influencing factors.

In order to reduce the number of entities n in a dataset, sampling or coresets [4] can be used to ultimately reduce the runtime of a clustering algorithm. Similar observations apply to (i) dimensionality reduction techniques, e.g., PCA or SVD, (ii) embeddings, or (iii) sketches, which all together aim to reduce the number of dimensions d of the dataset [1].

Regarding the number of clustering iterations ω, there are three categories of related work, which address the internals of a clustering iteration: (a) It has been shown that the initialization of k-center clustering algorithms is crucial to reduce the number of clustering iterations [5, 10]. The initial centroids of state-of-the-art initialization techniques are close to their optimum position, therefore requiring less clustering iterations until convergence. (b) Several works address how a single clustering iteration can be accelerated, i.e., by making distance calculations faster [7] or by caching previously calculated distances [12]. (c) Undoubtedly, reducing ω is crucial since it subsumes approaches from (a) and (b). Therefore, the "check for convergence" step is of paramount interest when reducing ω and thereby reducing the runtime of the clustering algorithm.

As each iteration comprises costly distance calculations, it is practically not feasible to perform a clustering algorithm on large datasets until convergence due to the excessive runtime. Hence, the question arises when to terminate the algorithm earlier than convergence. An easy approach to reduce the runtime of the clustering algorithm is to allow a fixed number of clustering iterations. However, it is challenging to choose a promising value for this threshold: Too few iterations lead to an imprecise result, whereas too many iterations lead to a high runtime. A generic threshold for all datasets is not feasible, because of too many influencing factors, such as the feature space or data distribution.

3 META-LEARNING TERMINATION (MTL)

In a previous work [8], we proposed a generic meta-learning approach to terminate k-center clustering algorithms early based on an arbitrary definable qualitative demand $q \in [0; 1]$. This approach relies on a correlation between the quality of intermediate clustering results throughout several clustering iterations and corresponding clustering validity measures (CVMs). To this end, the meta-learning procedure requires an offline phase, which gathers the necessary meta-knowledge, and an online phase, which applies the meta-knowledge on previously unseen datasets.

In the offline phase, a clustering algorithm is executed with varying parameter values on datasets with different characteristics. Throughout these executions, values of selected CVMs are recorded for each clustering iteration. The resulting values of these CVMs are often not normalized, i.e., the value ranges can be unbounded. In order to make these values tangible for analysts, we introduced the notion of quality $q \in [0; 1]$. That is, q indicates the quality of the intermediate clustering result after a certain clustering iteration in contrast to the quality of the last clustering iteration, which only becomes available after convergence. Hence, the clustering quality for each clustering iteration can only be investigated in retrospective after convergence. We proposed to create a correlation between the quality q and the CVMs, e.g., with a regression function that is trained during the offline phase based on several executions.

In the online phase, the analyst defines the expected qualitative demand q for the clustering result of a previously unseen dataset. Subsequently, the above-mentioned correlation is exploited to terminate the clustering algorithm earlier than convergence, while aiming to achieve the desired qualitative demand.

We showed that considerable runtime savings are possible, while regularly meeting the qualitative demand for several CVMs. However, we face two pitfalls regarding this method: (1) In order to exploit the correlation, it is first necessary to perform the offline phase. Since the offline phase comprises several executions of a k-center clustering algorithm, the high runtimes are solely moved from the online phase to the mandatory offline phase. (2) Creating a sound correlation between the quality and the corresponding CVM is an optimization problem. Influencing factors are for example datasets and their characteristics, the selected CVM, or chosen parameter values in the offline phase. Furthermore, formalizing the correlation itself, i.e., choosing a promising correlation method, poses another optimization problem.

To the best of our knowledge, this proposed approach is nonetheless currently the only one to limit the number of clustering iterations ω based on an arbitrarily predefinable qualitative demand. In the next section, we propose a novel method to reduce the number of clustering iterations based on a qualitative demand, which avoids the two mentioned pitfalls, yet still provides a tangible notion of quality for analysts.

4 GENERIC QUALITATIVE APPROXIMATION TERMINATION (GQA)

Before detailing on our new approach, we briefly summarize the basics of k-center clustering algorithms. Let $\mathcal{X}$ be a dataset with n entities and d dimensions, i.e., $\mathcal{X} \subset \mathbb{R}^d$. The goal of k-center clustering algorithms is to group $\mathcal{X}$ into k disjoint clusters, such that each entity is assigned to the closest centroid $c \in C$. As this problem is NP-hard [2], several heuristics exist, which aim to approximate the solution. One of these heuristics is the k-Means algorithm [16]. The goal of k-Means is to find the set C of k centroids which minimizes the objective function in Equation 1.

$$\phi_{\mathcal{X}}(C) = \sum_{x \in \mathcal{X}} \min_{c \in C} \|x - c\|^2 \tag{1}$$

Here, the Euclidean distance from an entity $x \in \mathcal{X}$ to the closest centroid $c \in C$ is calculated. $\phi_{\mathcal{X}}(C)$ denotes the sum of these distances over all entities in $\mathcal{X}$ and is also called sum of squared errors (SSE) for k-Means or variance for all k-center clustering algorithms. k-center algorithms minimize their notion of variance by moving these k centroids to a better position in each clustering iteration until a certain convergence criterion is met.

4.1 Intuition of our Approach

The goal of our approach is to exploit a tangible qualitative demand q in order to terminate the clustering algorithm as soon

as q is met. In contrast to MTL [8], we explicitly aim for no preparations, e.g., no prior meta-learning step.

Our novel approach GQA draws on two properties, which are valid independent of datasets and eventualities of k-center clustering algorithms, thus preserving generality.

Property 1: Monotonically decreasing variance. According to the objective function in Equation 1, the variance ϕ decreases throughout each clustering iteration, which is applicable for all k-center clustering algorithms. Manning et al. discuss this property for k-Means in detail [17], which can be transferred to other k-center clustering algorithms analogously.

Since ϕ is monotonically decreasing, we derive that the quality of the clustering is becoming better in each iteration (cf. Equation 1). Hence, we can formulate the gain in quality as changes of the variance between two subsequent iterations. To this end, we focus on the quotient σ_i of the variance between two subsequent clustering iterations $i-1$ and i, i.e., $\sigma_i = (\phi_{i-1}/\phi_i)$. Finally, k-center clustering algorithms converge as soon as $\phi_{i-1} = \phi_i$, hence $\sigma_i = 1$, i.e., the variance cannot be reduced any further. As σ_i typically becomes smaller per iteration and since we do not make any further assumptions on the dataset $\mathcal{X}$ and its variance in order to preserve generality, we can conclude that $\sigma_i \in [1; \infty]$.

Property 2: Notion of quality. Similarly to MTL, we also draw on a qualitative demand $q \in [0; 1]$ of the approximated clustering result, yet in a different way. That is, q can be specified by an analyst, where the choice of q has impact on the clustering result: If a very accurate clustering result is of interest, q should be set larger than for exploratory purposes, where already potentially moderate results may lead to valuable insights.

Yet, the question remains how to combine the qualitative demand $q \in [0; 1]$ of the approximated clustering result with $\sigma_i \in [1; \infty]$, while avoiding time-consuming preparations. Putting both above-mentioned properties together, we can formalize our approach as follows: During each clustering iteration, σ_i (and therefore ϕ_i) has to be calculated. Subsequently, terminate the k-center clustering algorithm as soon as Inequality 2 is satisfied.

$$1 - q \geq \sigma_i - 1 \tag{2}$$

Inequality 2 denotes that further clustering iterations would typically reduce $\sigma_i - 1$ less than $1 - q$. Note, that our approach preserves generality, since both properties are generally valid regarding dataset characteristics or k-center clustering algorithms.

4.2 Algorithm

Algorithm 1 outlines how GQA can be incorporated in the generic procedure of k-center clustering algorithms. The algorithm proceeds in four steps: (1) The centroids are initialized in line 1 according to a specific initialization, e.g., random, or k-Means‖ [5]. (2) The clustering is improved, i.e., the entities are assigned to the closest centroid (cf. lines 4-6). (3) The centroids are moved to the center of the cluster. To this end, the new position of the centroids $c \in C$ and the corresponding variance for each cluster are determined according to an objective function (cf. line 9). (4) Finally, the algorithm converges in line 16.

Changes in contrast to the regular procedure of k-center clustering algorithms are depicted underlined. As the variance for each single cluster is calculated in line 9, the overall variance for the current iteration ϕ_i is the sum of these individual values. Subsequently, σ_i is calculated in line 13, if a previous iteration was already performed. It is necessary to check this state, since our approach draws on the change of ϕ between two subsequent

Algorithm 1: k-center clustering algorithms with GQA.

Input: $\mathcal{X}$ - dataset, k - number of clusters, q - qualitative demand
Output: $\mathcal{K}$ - a combination ($\circ$) between $\mathcal{X}$ and the assigned centroid for each entity

```
/* initialize centroids                                          */
```
1 $C \leftarrow$ initialize a set of k centroids;
2 $i \leftarrow 0$;
3 **repeat**
```
    /* improve clustering                                        */
```
4 **for** $\forall x \in \mathcal{X}$ **do**
5 $\mathcal{K} \leftarrow \{x \circ c\}$, where c denotes the closest centroid to x according to an objective function;
6 **end**
7 $\phi_i(C) \leftarrow 0$;
```
    /* change centroids                                          */
```
8 **for** $\forall c \in C$ **do**
9 $c, \phi(c) \leftarrow$ new c and its corresponding variance according to an objective function, where $\{x \circ c\} \in \mathcal{K}$;
10 $\phi_i(C) \leftarrow \phi_i(C) + \phi(c)$;
11 **end**
12 **if** $i > 0$ **then**
13 $\sigma_i \leftarrow \phi_{i-1}(C)/\phi_i(C)$;
14 $\phi_{i-1}(C) \leftarrow \phi_i(C)$;
15 $i \leftarrow i + 1$;
16 **until** $i > 1$ *and* $1 - q \geq \sigma_i - 1$; `// check for convergence`
17 **return** $\mathcal{K}$;

iterations. After σ_i is calculated, ϕ is adjusted properly for the next iteration, i.e., ϕ of the previous iteration is set as ϕ of the current iteration. Finally, the convergence criterion is set according to Inequality 2 in line 16. Note however, that this convergence criterion can only be met after at least 2 clustering iterations are performed, since σ_i draws on the variance ϕ of two subsequent iterations. Hence, we introduce the additional check for $i > 1$ in the convergence criterion in line 16.

4.3 Analysis and Discussion

The goal of our approach is to keep additional calculations as cheap as possible. As already mentioned, the computations required by our approach are underlined in Algorithm 1. The computations rely on (a) allocations of variables (lines 7, 10, 13 and 14), (b) comparisons (lines 12 and 16), as well as (c) arithmetic operations (lines 10, 13, 16). Allocations and comparisons can be performed in $O(1)$. For the arithmetic operations, we reuse byproducts of the k-center clustering algorithm, such as ϕ_i. These values are used throughout several iterations with simple additions (line 10) or divisions (line 13). Therefore, the runtime complexity for the arithmetic operations is $O(1)$. Concluding, the overall runtime complexity of our approach is $O(1)$, thus preserving the runtime complexity of a k-center clustering algorithm without a significant runtime overhead.

In contrast to MTL, our novel approach GQA (i) can be used for ad-hoc exploratory clustering analyses, since it does not rely on a time-consuming meta-learning step, (ii) preserves generality regarding dataset characteristics, which may not be the case for MTL, since it obeys an optimization problem (cf. Section 3), and (iii) directly addresses the objective function of k-center clustering algorithms instead of additional CVMs as MTL does. As discussed above, addressing the objective function can be

Dataset	n	d	c
I - III	10,000	10	{10; 50; 100}
IV - VI	10,000	50	{10; 50; 100}
VII - IX	10,000	100	{10; 50; 100}
X - XII	100,000	10	{10; 50; 100}
XIII - XV	100,000	50	{10; 50; 100}
XVI - XVIII	100,000	100	{10; 50; 100}
XIX - XXI	1,000,000	10	{10; 50; 100}
XXII - XXIV	1,000,000	50	{10; 50; 100}
XXV - XXVII	1,000,000	100	{10; 50; 100}

Table 1: 27 synthetic datasets for the evaluation.

Dataset	Name	n	d	c
i	Skin segmentation	245,057	3	2
ii	Poker hand	1,025,010	10	10
iii	Individual household electric power consumption	2,049,280	7	-
iv	US census data (1990)	2,458,285	68	-
v	KDD Cup 1999 data	4,898,431	33	23
vi	SUSY	5,000,000	18	2
vii	Gas sensor array under dynamic gas mixtures	8,386,765	19	-
viii	HEPMASS	10,500,000	28	2
ix	HIGGS	11,000,000	28	2
x	Heterogeneity activity recognition	33,741,500	5	6

Table 2: 10 real-world datasets as used in the work of MTL [8], where c denotes #classes, if available.

done very efficiently, whereas using an additional CVM in each clustering iteration may require noticeable runtime overhead [8].

Furthermore, it should be emphasized that GQA can be easily used by analysts and automated exploration methods due to its striking resemblance to the regular procedure of k-center clustering algorithms. Because solely q should be defined, the additional complexity of our novel approach is clearly manageable.

5 EVALUATION

In our evaluation, we investigate the benefits of our proposed approach for exploratory clustering analyses. To this end, we will compare our novel approach GQA with its closest competitor MTL [8] and a regular execution of a k-center clustering algorithm. We present the setup for our evaluation, before presenting the runtime and quality results.

5.1 Experimental Setup

Hardware and Software. We conducted all of our experiments on a distributed Apache Spark cluster. This cluster consists of one master node and six worker nodes. The master node has a 12-core CPU with 2.10 GHz each and 192 GB RAM. Each worker has a 12-core CPU with 2.10 GHz each and 160 GB RAM. Each node in this cluster operates on Ubuntu 18.04. We installed OpenJDK 8u191, Scala 2.11.12, Hadoop 3.2.0 as well as Spark 2.4.0.

Synthetic Datasets. We implemented a synthetic dataset generator in order to perform a systematic evaluation with controlled dataset characteristics. This tool generates datasets based on the following input parameters: The number of entities in the dataset (n), the number of dimensions (d) and the number of clusters in a dataset (c), where each cluster contains n/c entities. Our tool generates datasets with values that lie within the range $[-10; 10]$ for each dimension. Each cluster has a Gaussian distribution with the mean at the center and a standard deviation of 0.5. The c centers are randomly chosen and the clusters are non-overlapping. Table 1 depicts the characteristics of the 27 synthetic datasets.

Real-World Datasets. We use the same datasets as in our prior work for MTL [8], which are publicly available[3]. Table 2 summarizes the characteristics of these 10 datasets. Note, that not all of them have class labels, i.e., the number of classes is unknown for some datasets (indicated by "-" in Table 2).

Implementation. We base our implementation on Apache Spark. We focus on k-Means as instantiation of a k-center clustering algorithm due to its overwhelming popularity [18]. To this end, we implemented several methods to terminate k-Means.

[3] https://archive.ics.uci.edu/ml/datasets.php

The baseline (BASE) for this experiment is Spark's MLlib implementation of k-Means. This implementation uses k-Means‖ [5] for the initialization step and terminates after 20 clustering iterations at most. We explicitly remove the threshold for the number of clustering iterations, since we want to highlight the differences in quality when performing k-Means until convergence.

For MTL, it should be noted that this obeys an optimization problem regarding (i) the used datasets and their characteristics, and (ii) the choice of the used correlation technique. However, for synthetic datasets, we have closely followed the approach described in our previous work [8]. For the offline phase, we used datasets from Table 1 where $d = 50$. We clustered these datasets with k in 11 equidistant values in $[2; 2c]$ and let each clustering run until Spark's convergence criterion is met. We performed three runs per value of k. For each clustering iteration, we measured the SSE as well as the separation (SEP) between the centroids, since we achieved the best results with the SEP metric in our previous work [8]. Subsequently, we trained a second-degree polynomial regression between the change rate of SEP in contrast to the relative error of the SSE regarding the current clustering iteration and the final clustering iteration. The whole process took 11.17 hours and is thus not feasible for ad-hoc exploratory clustering analyses. Regarding the real-world datasets, we use the same ones as in [8], i.e., we also use the same regression function. The meta-learning process on those real-world datasets required several days, which emphasizes the impractical runtime for the offline phase of MTL. Furthermore, we implemented our novel GQA approach as described in Section 4.

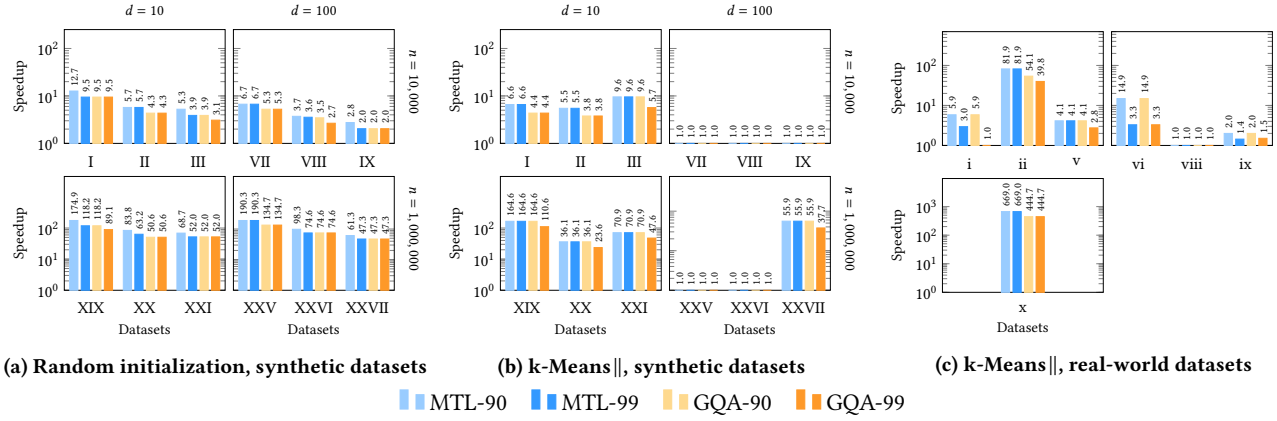
For MTL and GQA, we set the respective qualitative demands to 90 % and 99 % in order to achieve valuable results. Hence, we compare Spark's implementation (BASE) to MTL-90, MTL-99, GQA-90 and GQA-99 on synthetic and real-world datasets.

Furthermore, we use different initialization techniques in order to investigate the differences in the results. We run k-Means with $k = c$ for each dataset, where c is known. For each method, we performed three runs and present median values.

5.2 Runtime Results

Figure 1 summarizes the results regarding the speedups in contrast to the baseline, where Figure 1a focuses on the results with random initialization and Figure 1b addresses the results with the initialization via k-Means‖. While both of them show the results on synthetic datasets, Figure 1c unveils the results with k-Means‖ initialization on real-world datasets.

It can be seen that for random initialization, speedups can be achieved for all datasets in contrast to the initialization via k-Means‖. Since k-Means‖ initializes centroids closer to their optimum, less clustering iterations are necessary than for random



(a) Random initialization, synthetic datasets **(b) k-Means‖, synthetic datasets** **(c) k-Means‖, real-world datasets**

Figure 1: Speedup of MTL and GQA in contrast to BASE for all synthetic and real-world datasets where $k = c$.

initialization. Hence, MTL and GQA lead to remarkable speedups for random initialization of up to 190.3 (MTL) and 134.7 (GQA).

On the other hand, k-Means‖ provides an $O(log\ k)$-approximation to the final clustering result [5]. Therefore, some speedups of 1 are observed, i.e., BASE terminates k-Means as early as MTL and GQA. However, as shown in Figure 1b, MTL and GQA still provide strong speedups in many cases, such as for datasets XIX and XXVII. We observe speedups of roughly up to 164.6. When comparing dataset IX with dataset XXVII, it is evident that the latter has 100× more entities, where the remaining dataset characteristics, such as d and c, remain unchanged. As k-Means‖ does not select promising initial centroids for dataset XXVII, several clustering iterations are necessary for BASE. Yet, MTL and GQA terminate k-Means earlier than convergence, thus achieving significant speedups. More generally speaking, k-Means‖ can only select promising initial centroids, if the underlying dataset characteristics and number of clusters are a strong fit for its procedure.

Regarding the results of the real-world data (cf. Figure 1c), we make very similar observations. Here, even more significant speedups of up to 669.0 (MTL) and 444.7 (GQA) can be observed for the largest real-world dataset x.

In general, MTL and GQA achieve similar speedups, however MTL is faster in some cases. Note, that MTL addresses the SEP, whereas GQA addresses the variance (= SSE for k-Means) in order to approximate clustering results. Since the SEP typically changes more significantly in the first few iterations than the SSE, we argue that GQA requires a few additional clustering iterations.

Regarding the different qualitative demands, we observe only small deviations in the resulting speedups. These differences mostly occurred on real-world datasets (cf. Figure 1c). Here, the higher qualitative demand of 99 % requires more iterations and therefore leads to lower speedups.

We conclude that both approaches are well-suited for rather voluminous datasets, since k-center clustering algorithms require more clustering iterations on these datasets until convergence, which is explicitly addressed by MTL and GQA.

5.3 Quality Results

Since MTL and GQA are able to speedup the execution of a clustering algorithm significantly, the question remains how these approximations affect the clustering quality.

As clustering aims to provide compact and well-separated clusters, we focus on the compactness and the separation (SEP) of the centroids. We use the sum of squared errors (SSE) as instantiation of the compactness, since the smaller the SSE for k-Means, the less variance in the clusters, i.e., the more compact are the clusters. For a better comparison, we focus on the relative error δ of SSE and SEP of the clustering results of MTL and GQA in contrast to the baseline, i.e., the smaller the better.

Figure 2 summarizes the results for MTL and GQA. Note the different y-axes throughout the figures. The results unveil that the qualitative demands of 90 % and 99 % are mostly met in average for MTL and GQA. It should be noted, that the qualitative demand of GQA addresses the SSE (left), whereas the qualitative demand of MTL addresses the SEP (right). A more detailed investigation of the results unveiled that for all synthetic datasets, the respective qualitative demands are always met. Regarding the real-world datasets, MTL and GQA rarely terminated the clustering algorithm too early, thus omitting better clustering results. This happened for both approaches only once for a qualitative demand of 90 % and twice for a qualitative demand of 99 %. This observation supports the practical feasibility of our novel approach GQA, i.e., addressing the decreasing trend of σ_i leads to satisfying clustering results in practice.

The quality of the clustering results achieved by MTL and GQA differ only marginally from the baseline, i.e., the regular execution of k-Means. Furthermore, the initialization via k-Means‖ leads to more compact and better separated clustering results than initializing at random, since the values for δSSE and δSEP are mostly smaller (cf. Figure 2a and b). Hence, this initialization is better suited for exploratory clustering analysis, since it allows more correct insights into clustering results.

Moreover, the results achieved by GQA are always better than the corresponding pendant of MTL in terms of δSSE and δSEP. The reason can be found in the additional clustering iterations that GQA performed in contrast to MTL (cf. Section 5.2).

Concluding, MTL and GQA perform similar in terms of runtime, yet GQA does not rely on a previously conducted offline phase for meta-learning. Remember, that MTL required several executions of a k-center clustering algorithm on several datasets, which required more than 11 hours for synthetic datasets and several days for real-world datasets in our scenario. In addition, MTL and GQA lead to compact and well-separated clustering results. However, GQA achieves better clustering results, because

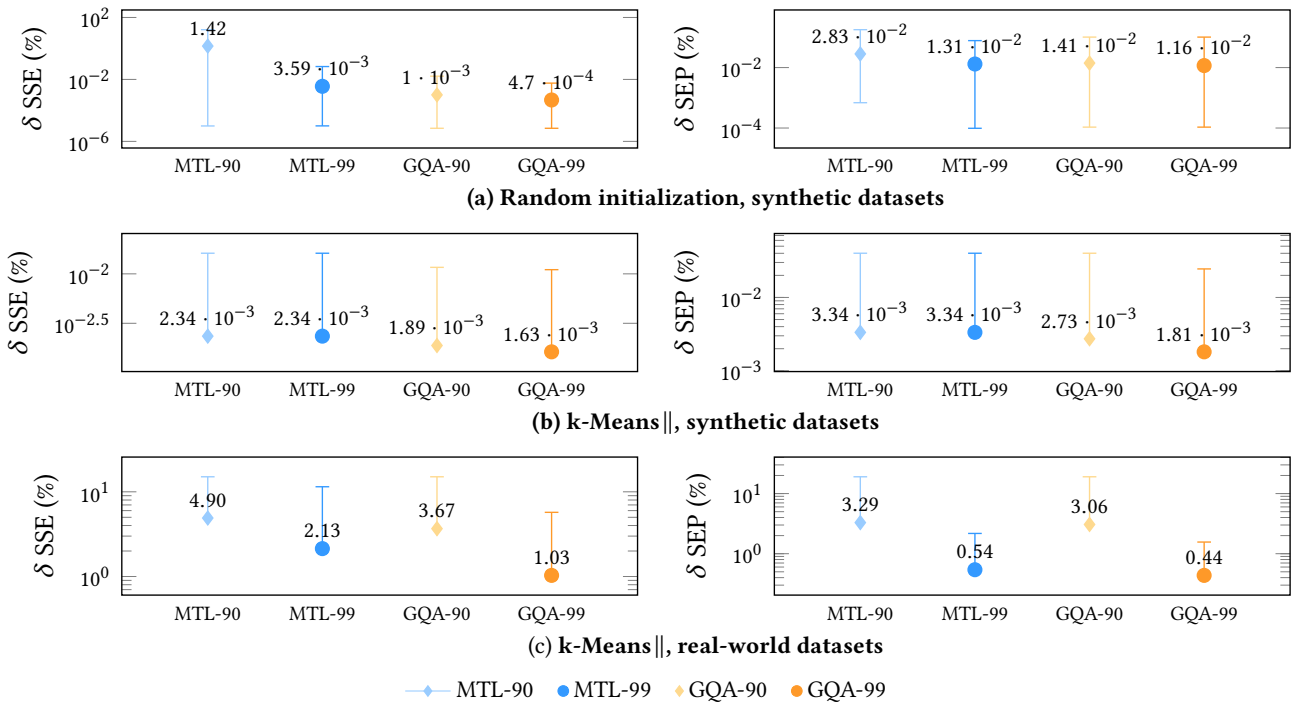


Figure 2: Relative error δ of SSE and SEP compared to the baseline. Average values are depicted per error bar. Smaller values indicate more similar clustering results w.r.t. the baseline, i.e., they show better clustering results.

it typically requires a few more clustering iterations than MTL. Therefore, our novel approach GQA is well-suited for ad-hoc exploratory clustering analyses, especially on voluminous datasets, since it provides very accurate results in a short time period.

6 CONCLUSION

In this work, we proposed a novel approach to terminate k-center clustering algorithms as soon as a predefined qualitative demand of the clustering results is met. Our approach aims to trade off quality of clustering results, while achieving them in a short time frame. We showed that our approach is generic, i.e., it can be used with several k-center clustering algorithms and initialization strategies. In our comprehensive evaluation, we unveiled that our approach significantly outperforms state-of-the-art executions of k-center clustering algorithms in terms of runtime, yet achieves very similar clustering results. Therefore, it is of particular interest for exploratory clustering analyses. Future work will address to what extent automated exploration methods benefit from our novel approach.

ACKNOWLEDGMENTS

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program 'Services Computing'. Some work presented in this paper was performed in the project 'INTERACT' as part of the Software Campus program, which is funded by the German Federal Ministry of Education and Research (BMBF) under Grant No.: 01IS17051.

REFERENCES

[1] Amirali Abdullah, Ravi Kumar, Andrew McGregor, Sergei Vassilvitskii, and Suresh Venkatasubramanian. 2016. Sketching, embedding, and dimensionality reduction for information spaces. In *AISTATS 2016*, Vol. 41. 948–956.

[2] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. 2009. NP-hardness of Euclidean sum-of-squares clustering. *Machine Learning* 75, 2 (may 2009), 245–248.

[3] David Arthur and Sergei Vassilvitskii. 2006. How slow is the k-means method?. In *Proceedings of the Annual Symposium on Computational Geometry*.

[4] Olivier Bachem, Mario Lucic, and Andreas Krause. 2018. Scalable k-means clustering via lightweight coresets. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1119–1127.

[5] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Scalable K-Means++. *PVLDB* 5, 7 (2012), 622–633.

[6] P S Bradley, O L Mangasarian, and W. N. Street. 1997. Clustering via concave minimization. In *Advances in Neural Information Processing Systems*. 368–374.

[7] Charles Elkan. 2003. Using the Triangle Inequality to Accelerate k-Means. *International Conference on Machine Learning* (2003), 147–153.

[8] Manuel Fritz, Michael Behringer, and Holger Schwarz. 2019. Quality-driven early stopping for explorative cluster analysis for big data. *SICS Software-Intensive Cyber-Physical Systems* 34, 2-3 (jun 2019), 129–140.

[9] Manuel Fritz, Michael Behringer, and Holger Schwarz. 2020. LOG-Means: Efficiently Estimating the Number of Clusters in Large Datasets. *PVLDB* 13, 11 (2020), 2118 – 2131.

[10] Manuel Fritz and Holger Schwarz. 2019. Initializing k-means efficiently: Benefits for exploratory cluster analysis. In *Lecture Notes in Computer Science*, Vol. 11877 LNCS. Springer, 146–163.

[11] Manuel Fritz, Dennis Tschechlov, and Holger Schwarz. 2020. Learning from past observations: Meta-learning for efficient clustering analyses. In *Lecture Notes in Computer Science*, Vol. 12393 LNCS. Springer, 364–379.

[12] Greg Hamerly. 2010. Making k-means even faster. In *Proceedings of the 2010 SIAM international conference on data mining*. 130–140.

[13] Greg Hamerly and Jonathan Drake. 2015. Accelerating Lloyd's Algorithm for k-Means Clustering. In *Partitional Clustering Algorithms*. Springer, 41–78.

[14] Zhexue Huang. 1997. A Fast Clustering Algorithm to Cluster Very Large Categorical Data Sets in Data Mining. *Research Issues on Data Mining and Knowledge Discovery* (1997), 1–8.

[15] Anil K. Jain. 2010. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters* 31, 8 (jun 2010), 651–666.

[16] James B. Macqueen. 1967. Some methods for classification and analysis of multivariate observations. *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability* 1 (1967), 281–297.

[17] Christopher D. Manning, Prabhakar. Raghavan, and Hinrich. Schütze. 2008. *Introduction to information retrieval.* Cambridge University Press. 482 pages.

[18] Xindong Wu, Vipin Kumar, Quinlan J. Ross, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S Yu, Zhi Hua Zhou, Michael Steinbach, David J Hand, and Dan Steinberg. 2008. Top 10 algorithms in data mining. *Knowledge and Information Systems* 14, 1 (2008), 1–37.

Short Paper

AutoML4Clust: Efficient AutoML for Clustering Analyses

Dennis Tschechlov, Manuel Fritz, Holger Schwarz
University of Stuttgart
Stuttgart, Germany
{dennis.tschechlov,manuel.fritz,holger.schwarz}@ipvs.uni-stuttgart.de

ABSTRACT

Data analysis is a highly iterative process. In order to achieve valuable analysis results, analysts typically execute many configurations, i.e., algorithms and their hyperparameter settings, based on their domain knowledge. While experienced analysts may be able to define small search spaces for promising configurations, especially novice analysts define large search spaces due to their lack of domain knowledge. In the worst case, they perform an exhaustive search throughout the whole search space, resulting in infeasible runtimes. Recent advances in the research area of AutoML address this challenge by supporting novice analysts in the combined algorithm selection and hyperparameter optimization (CASH) problem for supervised learning tasks. However, no such systems exist for unsupervised learning tasks, such as the prevalent task of clustering analysis. In this work, we present our novel AutoML4Clust approach, which efficiently supports novice analysts regarding CASH for clustering analyses. To the best of our knowledge, this is the first thoroughly elaborated approach in this area. Our comprehensive evaluation unveils that AutoML4Clust significantly outperforms several existing approaches, as it achieves considerable speedups for the CASH problem, while still achieving very valuable clustering results.

1 INTRODUCTION

Data analysis is a crucial discipline to extract knowledge and insights from data. Therefore, analysts apply data mining techniques, typically machine learning algorithms, to extract patterns from data and to gain insights about data. A fundamental primitive in data mining is clustering analysis, which is an unsupervised machine learning task being used in various application domains, e.g., computer vision, document clustering, for business purposes, or to study genome data in biology [11].

Throughout these manifold fields of application domains, analysts typically struggle with the selection of a promising clustering configuration, i.e., a clustering algorithm and its corresponding hyperparameter settings, that achieves valuable clustering results. Hence, analysts typically define a configuration space, i.e., a search space of clustering algorithms and their hyperparameter settings, in which they expect promising configurations. Yet, novice analysts lack in-depth domain knowledge and hence define very large configuration spaces. In the worst case, novice analysts cannot limit configuration spaces at all and perform an exhaustive search throughout all possible configurations. Since this exhaustive search is very time-consuming, novice analysts often explore only a few configurations, e.g., randomly selected, from the configuration space, which often leads to solely moderate results. Hence, novice analysts require support to achieve valuable clustering results in a reasonable amount of time.

For supervised learning tasks, recent advances in the research area of AutoML are able to support novice analysts in the combined algorithm selection and hyperparameter optimization (CASH) problem in an automated and efficient manner [5, 18]. These approaches greedily explore large configuration spaces by trading off exploration and exploitation strategies, thus avoiding a time-consuming or even infeasible exhaustive search.

In this work, we propose AutoML4Clust, an efficient AutoML approach to support novice analysts in the CASH problem for clustering analyses. To the best of our knowledge, this is the first thoroughly elaborated AutoML approach for efficient clustering analyses, capable of automatically selecting promising clustering algorithms and their hyperparameters in combination.

Our contributions include the following:

- We introduce AutoML4Clust, our novel approach to efficiently support novice analysts in the prevalent CASH problem for clustering analyses.
- We reveal that AutoML4Clust is generic, i.e., it can be instantiated with different AutoML concepts, clustering algorithms and clustering metrics.
- In our evaluation, we unveil that AutoML4Clust significantly outperforms several existing approaches, as it achieves speedups of up to 437x for the CASH problem, while still achieving valuable clustering results. Hence, AutoML4Clust efficiently supports novice analysts in the CASH problem for clustering analyses.

The remainder of this paper is structured as follows: We present related work in this area in Section 2. In Section 3, we present AutoML4Clust, our novel AutoML approach for clustering analyses. In Section 4, we unveil the results of our evaluation. Finally, we conclude this work in Section 5.

2 RELATED WORK

Based on the generally accepted separation of machine learning tasks, we separate related work into support for the CASH problem regarding supervised and unsupervised learning tasks. We distinguish two important groups of related work to support analysts regarding the CASH problem: (1) AutoML systems for supervised learning, and (2) methods for unsupervised clustering analyses that either explore the algorithm selection or the hyperparameter optimization.

We define a configuration c as the combination of an algorithm $a \in \mathcal{A}$ and its hyperparameters $h \in \mathcal{H}$. Hence, we define the configuration space as $\mathcal{CS} = \mathcal{A} \times \mathcal{H}$. In the following, we investigate related work based on the machine learning task and its ability to explore $\mathcal{CS}$.

2.1 AutoML Systems for Supervised Learning

AutoML systems arose in the area of supervised machine learning to support novice analysts in the combined algorithm selection and hyperparameter optimization problem [5, 18]. As class labels are already available in the datasets, $\mathcal{CS} = \mathcal{A} \times \mathcal{H}$ can be explored automatically. The common underlying procedure of existing

AutoML systems is as follows: Given a budget, training data and an optimization metric, they execute and evaluate different configurations from $\mathcal{CS}$, i.e., they execute classification algorithms with the specified hyperparameter settings, and return the configuration that yields the best value for the optimization metric. In order to steer the exploration towards valuable results, several hyperparameter optimization techniques are proposed, such as Bayes [1], Hyperband [13], or BOHB [3]. These optimization techniques proceed in a greedy manner, since they define a specific trade-off between exploration, i.e., exploring new regions in the $\mathcal{CS}$, and exploitation, i.e., exploiting regions in the $\mathcal{CS}$, where already executed configurations performed well.

2.2 Algorithm Selection and Hyperparameter Optimization for Clustering Analyses

Related work that supports novice analysts regarding CASH for clustering analyses can be divided into methods that consider algorithm selection and methods that consider hyperparameter optimization.

Algorithm Selection. For unsupervised learning tasks, several approaches were developed to support novice analysts with the selection of a promising clustering algorithm, i.e., $\mathcal{CS} = \mathcal{A}$ for certain problems [4, 16]. These methods are based on meta-learning, i.e., they learn from past experiences in order to select the most promising algorithm on a previously unseen dataset.

Existing approaches differ in the used (a) implementation of the meta-learning steps, (b) clustering algorithms, and (c) metrics for evaluating the clustering results. However, these approaches solely focus on the clustering algorithm, yet completely ignore the corresponding hyperparameters $\mathcal{H}$.

Hyperparameter Optimization. Regarding the hyperparameter optimization, i.e., $\mathcal{CS} = \mathcal{H}$ of clustering algorithms, two types of approaches exist [7]: While exhaustive methods execute all configurations from $\mathcal{CS}$, non-exhaustive methods only execute some configurations. However, non-exhaustive methods are designed to optimize the hyperparameters of specific algorithms, e.g., k-Means. Exhaustive methods could also be applied for $\mathcal{CS} = \mathcal{A} \times \mathcal{H}$, though this results in tremendous runtime as the whole configuration space has to be explored.

Summary. Summarizing related work, existing AutoML systems only focus on supervised learning algorithms, yet can explore valuable results, where $\mathcal{CS} = \mathcal{A} \times \mathcal{H}$. For clustering analyses, there are approaches that either conduct an exploration for valuable clustering algorithms ($\mathcal{CS} = \mathcal{A}$) or their hyperparameters ($\mathcal{CS} = \mathcal{H}$). However, they do not address the combination of both, i.e., the CASH problem for clustering analyses, where $\mathcal{CS} = \mathcal{A} \times \mathcal{H}$, which is a crucial problem for novice analysts. We identified only some experimental implementations[1] [2] that address this problem, however they use (a) one specific optimization technique, or (b) one specific clustering metric, without explaining, evaluating or justifying their choice regarding (a) and (b). In addition, they miss a clear scientific elaboration and a systematic evaluation in comparison to existing approaches in this area.

3 AUTOML4CLUST

In this section, we introduce our generic AutoML4Clust approach to support novice analysts regarding the CASH problem for clustering analyses, where we apply concepts from existing supervised AutoML systems on clustering. Existing AutoML systems solely focus on supervised learning tasks, whereas we focus on clustering analysis, which is an unsupervised learning task. The key difference between supervised and unsupervised learning tasks is that the input datasets for unsupervised learning tasks do not contain ground-truth labels. Therefore, it is not possible to evaluate the result based on an external metric. Consequently, existing AutoML systems and their components cannot be applied per se for clustering analyses.

Figure 1 presents the procedure of our AutoML4Clust approach. Similar to supervised AutoML systems, it draws on a configuration space $\mathcal{CS}$, which defines the set of configurations that can be selected, executed and evaluated during the optimizer loop, which is at the core of existing hyperparameter optimization techniques. To this end, we rely on the configuration space $\mathcal{CS} = \mathcal{A} \times \mathcal{H}$. When considering different families of clustering algorithms, e.g., k-center and density-based ones, $\mathcal{CS}$ has to be defined in a hierarchical way, i.e., by defining the algorithm as conditional root-level hyperparameter [18]. Our AutoML4Clust procedure is structured into three parts (cf. Figure 1): The inputs, the optimizer loop, and return best configuration. In the following, we present these three parts in detail and subsequently discuss the benefits of AutoML4Clust.

3.1 Inputs

AutoML4Clust requires three inputs prior to execution. These are a dataset $\mathcal{D}$, an internal metric $\mathcal{M}$, and a budget l. Here, $\mathcal{D}$ does not contain any additional information, e.g., class labels. Hence, $\mathcal{M}$ is an internal metric that evaluates the internal structure of a clustering result. In the literature, many different internal metrics with different objectives are proposed [11, 14]. Most of them measure the compactness and the separation of clusters in different variations. Subsequently, they consider a quotient of both. The budget l defines the resources that can be used by the system. A common choice for the budget is a time constraint to limit the runtime or the number of configurations to execute. In this work, we use the number of optimizer loops that are performed as budget. However, we note that choosing an appropriate kind of budget and also an appropriate value for the budget is a difficult task. A too large value may lead to a long runtime, whereas a too small one can lead to solely moderate results.

3.2 Optimizer Loop

In the optimizer loop, an optimizer such as Random [1], Bayes [18], Hyperband [13], or BOHB [3] is used to find well-performing configurations efficiently. Here, the optimizer performs l loops, where each loop consists of three steps:

i) *Selection:* $c \in \mathcal{CS}$, where the optimizer selects a configuration c from the configuration space $\mathcal{CS}$. The different aforementioned optimizers mostly differ in their greedy procedure, i.e., trading off exploration and exploitation, in order to select a configuration $c \in CS$ in each optimizer loop l_i. Yet, all optimizers require the definition of a black-box function $f : \mathcal{CS} \rightarrow \mathbb{R}$, which is subject to optimization. This function f assigns each configuration $c \in \mathcal{CS}$ a metric value and is implemented with the following steps ii) and iii).

ii) *Execution:* $\mathcal{R} \leftarrow c(\mathcal{D})$. Here, the previously selected configuration c is executed on $\mathcal{D}$. The result of the execution is $\mathcal{R}$, which can be any kind of clustering result, e.g., the resulting labels or the final centroids.

[1] https://git.io/JUNKu [2] https://git.io/JUNKz

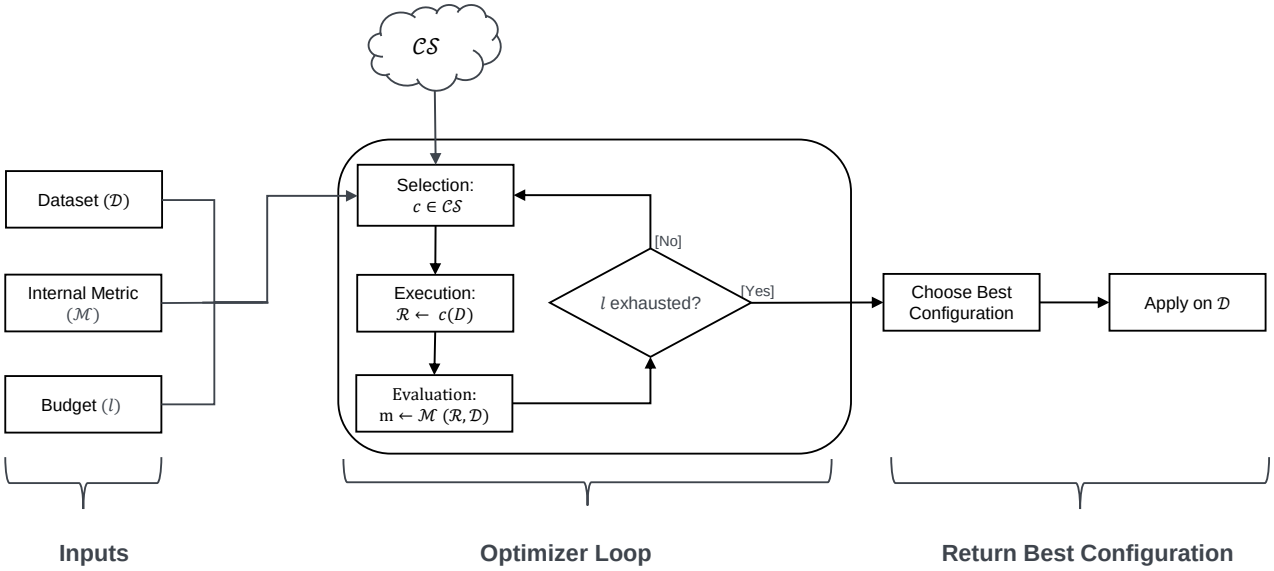


Figure 1: Procedure of AutoML4Clust.

iii) *Evaluation:* $m \leftarrow \mathcal{M}(\mathcal{R}, \mathcal{D})$, where the clustering result is evaluated. This means that the metric $\mathcal{M}$ is calculated based on the clustering result $\mathcal{R}$ and on the dataset $\mathcal{D}$.

3.3 Return Best Configuration

In the third step, the best configuration from all considered configurations is chosen. This is the configuration that achieves the best metric value regarding $\mathcal{M}$ from all configurations that are selected, executed and evaluated during each optimizer loop. This configuration is applied on the dataset $\mathcal{D}$ to finally obtain the best clustering result that is found by AutoML4Clust.

3.4 Discussion

Existing AutoML systems solely focus on supervised learning tasks in order to achieve valuable results, where $\mathcal{CS} = \mathcal{A} \times \mathcal{H}$, i.e., $\mathcal{CS}$ is typically very large. In contrast, AutoML4Clust addresses this problem for the prevalent unsupervised task of clustering analyses, where ground-truth labels are missing. Therefore, especially novice analysts are supported, which can neither limit $\mathcal{A}$ nor $\mathcal{H}$ to a manageable size and thus perform in the worst case an exhaustive exploration throughout $\mathcal{CS}$. Our proposed procedure draws on the latest fundamental concepts of existing AutoML systems, yet remains generic regarding the used clustering algorithms, metrics, and optimizers.

Regarding possible clustering algorithms, AutoML4Clust can use any kind of clustering algorithm by defining $\mathcal{CS}$ in a hierarchical way, similar to existing AutoML systems for supervised learning tasks. Regarding the metrics, AutoML4Clust draws on an internal metrics to asses the internal structure of a clustering result. Furthermore, several optimizers can be used, which follow the three steps (1) select a configuration, (2) execute the configuration, and (3) evaluate the result of the configuration.

Especially the combination of an internal metric and the used optimizer is of paramount importance: Since optimizers proceed in a greedy manner by defining a trade-off between exploration and exploitation throughout $\mathcal{CS}$, it is per se not clear which internal metric supports their behavior best. Yet, as prior work in the area of supervised learning has shown, these optimizers are able to efficiently explore large configuration spaces, while still achieving valuable results [3, 5, 18]. Therefore, we assume that AutoML4Clust similarly benefits from these optimizers. That is, we argue that AutoML4Clust is able to efficiently achieve valuable results within a predefined budget l. However, it is a very challenging task for novice analysts to specify such a suitable budget, since a too small budget leads to imprecise results, whereas a too large budget leads to long runtimes. Furthermore, it is not clear at all, if the used metric supports the greedy behavior of the optimizers within a reasonable budget.

4 EVALUATION

Since analysts are interested in fast and valuable results, the question remains how well different instantiations of our approach, i.e., combinations of optimizers and metrics, perform in order to achieve this goal. To this end, we compare in our evaluation how our novel AutoML4Clust approach performs (i) with different instantiations of optimizers and metrics for clustering analyses, and (ii) in contrast to existing approaches in this area. We first discuss the setup of our evaluation, before we investigate the accuracy of the results of different instantiations of AutoML4Clust in contrast to existing approaches. Subsequently, we analyze the runtime of AutoML4Clust in contrast to existing approaches. Finally, we show the practical feasibility of AutoML4Clust on real-world datasets regarding accuracy and runtime.

4.1 Setup

In the following, we describe the setup of our experiments. We focus on (i) the used hard- and software, (ii) the synthetic and real-world datasets that we use, (iii) the implementation details, and (iv) the performed experiment with its baselines.

Dataset	n	d	k_{act}
Statlog (Landsat Satellite)	6,435	35	7
ISOLET	7,797	617	26
Motion Capture Hand Postures	7,805	34	5
Avila	10,430	10	12
Pen-Based Recognition of Handwritten Digits	10,992	16	10

Table 1: Real-world datasets with their corresponding dataset characteristics n, d and k_{act}.

4.1.1 Hard- and Software. The experiments are performed on a virtual machine, which operates on Ubuntu 18.04. It has a 6-core CPU with 2.5 GHz and 32 GB RAM. Our implementation is based on Python 3.6 and on scikit-learn[3].

4.1.2 Datasets. We draw our evaluation on synthetically generated and real-world datasets. Regarding the synthetic datasets, we use the dataset generation tool from [7–9]. This tool generates datasets based on these four input characteristics:

1) n, which describes the number of entities, 2) d, which denotes the number of dimensions, where the values in each dimension lie in the interval $[-10, 10]$, 3) k_{act}, which is the actual number of clusters, where each cluster contains $\frac{n}{k_{act}}$ entities and 4) r, which is the ratio of noise, i.e., $\frac{r}{100} \cdot n$ additional entities are added uniformly at random to the dataset.

We generate datasets with $n \in [2{,}500; 7{,}500]$, $d \in [20; 40]$, $k_{act} \in [25; 75]$, and $r \in [0; 17; 50]$. We generate these datasets as a cross product of the above-mentioned characteristics, i.e., 24 synthetic datasets are used within our evaluation.

For the real-world datasets, we use 5 classification datasets from the UCI machine learning repository[4] with different dataset characteristics regarding n, d and k_{act}. Here, k_{act} describes the number of classes in the dataset. We removed the class labels from these datasets when applying instantiations of our AutoML4Clust approach and solely used them to evaluate the accuracy of our approach. In order to use these datasets for clustering, we removed any non-numeric and symbolic values, IDs, timestamps, class labels and empty values. Table 1 summarizes the datasets' characteristics. Note, that these datasets exhibit similar or even larger characteristics as the synthetic datasets regarding n and d.

4.1.3 Implementation. We evaluate AutoML4Clust with overall 12 instantiations, i.e., four optimizers and three internal metrics to unveil the best-performing combinations thereof. We provide our prototypical implementation of all AutoML4Clust instantiations in Python[5] with all versions of the used libraries[6].

Optimizers: We use the following four frequently used optimizers from the area of hyperparameter optimization (cf. Section 2.1): Random Search (RS) [1], Bayesian Optimization (BO) [1], Hyperband (HB) [13], and the combination of Bayesian Optimization and Hyperband (BOHB) [3]. We define the budget as number of optimizer loops that each optimizer performs.

Clustering algorithms: We focus on k-center clustering algorithms, due to their appealing runtime behavior and their popularity across researchers and practitioners [20]. We note that other kind of clustering algorithms, e.g., density-based ones like DBSCAN, have a runtime complexity of $O(n^2)$ or even higher, which makes them infeasible in practice for large datasets [10]. Therefore, we use k-Means [15], MiniBatch k-Means [17], k-Medoids [12], and GMM [2] as concrete instantiations of k-center clustering algorithms. We set the maximum number of clustering iterations for each algorithm to ten since Fritz et al. showed that even a few iterations already lead to valuable results [6].

Internal metrics: For the evaluation of the clustering result in each optimizer loop, we use three commonly used internal metrics that are implemented in scikit-learn: Calinski-Harabasz (CH), the Davies-Bouldin Index (DBI), and the Silhouettte (SIL).

4.1.4 CASH Experiment and Baselines. Based on $\mathcal{CS} = \mathcal{A} \times \mathcal{H}$, we define the CASH experiment analogue to related work in the area of AutoML [18]. We set $\mathcal{A}$ as described in Section 4.1.3. We set the search space $\mathcal{H}$ for the hyperparameter k of k-center clustering algorithms to $\mathcal{H} = \{2, \ldots, \frac{n}{10}\}$, i.e., the maximum k value is set in relation to the number of entities in the dataset. Since analysts perform in the worst-case an exhaustive search throughout $\mathcal{CS}$ due to the lack of more efficient approaches, we compare AutoML4Clust to an exhaustive search.

4.2 Accuracy Evaluation

In this section, we investigate the accuracy obtained by different instantiations of AutoML4Clust in contrast to the baselines. Therefore, we explain how we (i) examine a suitable budget to achieve valuable clustering results, (ii) compare the accuracy with the baselines, and (iii) discuss the effect of noisy data.

Since the actual labels of the datasets are known in our experiments, we use an external clustering metric to asses the accuracy of the achieved clustering result, similar to the accuracy from classification tasks. Therefore, we use the adjusted mutual information (AMI) [19], which is limited to $[0; 1]$, while values closer to one indicate a better matching of the predicted labeling with the actual labeling of the dataset.

4.2.1 Time to Accuracy. Figure 2 summarizes the accuracy results of the AutoML4Clust instantiations, i.e., the four optimizers and the three internal metrics at each optimizer loop l_i.

Budget: After about 60 optimizer loops, i.e., $l_i = 60$ (which is marked by the vertical line), the accuracy of AutoML4Clust does not further improve significantly for all instantiations. Hence, we argue that $l = 60$ is a suitable budget for AutoML4Clust to support novice analyst in achieving valuable results efficiently.

AutoML4Clust accuracy: AutoML4Clust achieves with every optimizer very accurate results, i.e., AMI values over 90%. Furthermore, we observe that more optimizer loops increase the accuracy.

AutoML4Clust instantiations: The AutoML4Clust instantiations with the CH metric achieve the highest accuracy. The reason for this is that the CH metric focuses on the intra- and inter-cluster compactness. Therefore, in contrast to DBI, it is less sensitive to sub-clusters, i.e., two or more clusters in a dataset that are very close to each other [14]. The most inaccurate results are obtained by instantiations with the SIL metric. This can be explained by the calculation of the SIL, as it can be highly influenced by the position of single entities. Due to its imprecise results, we do not present results for the SIL metric in the remaining experiments.

4.2.2 Accuracy Comparison. Figure 3 unveils the accuracy of AutoML4Clust in comparison to the respective baselines. We present the results of the AutoML4Clust instantiations with the four optimizers, the budget of $l = 60$, and the CH and DBI metrics.

[3] scikit-learn.org [4] https://archive.ics.uci.edu/ml/datasets.php
[5] https://git.io/JTeix [6] https://git.io/JTeXG

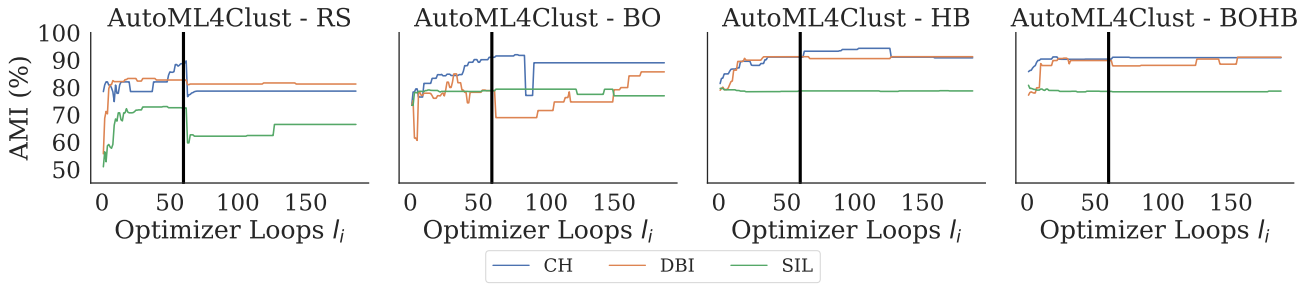


Figure 2: Accuracy of the AutoML4Clust instantiations over all synthetic datasets at each optimizer loop l_i for the CASH experiment. The vertical line at $l_i = 60$ marks where the accuracy does not further improve significantly.

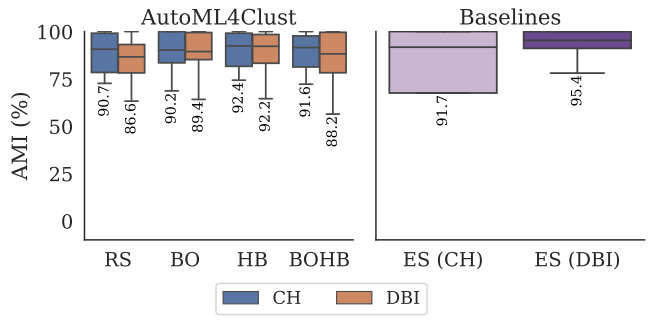


Figure 3: Accuracy of AutoML4Clust over synthetic datasets in contrast to exhaustive search (ES) with CH and DBI. Median values are shown at each box plot.

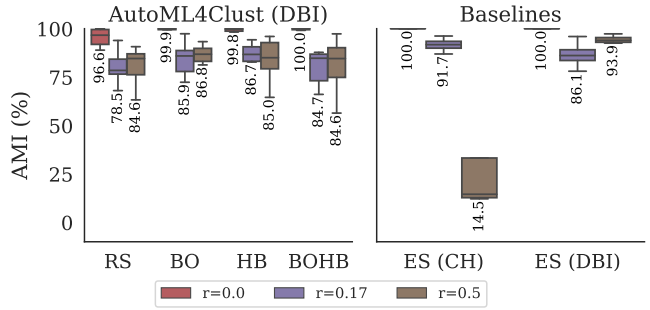


Figure 4: Comparison of the impact of noise (r) for AutoML4Clust and the baselines regarding the AMI results. Median values are shown at each box plot.

AutoML4Clust achieves similarly accurate results as the exhaustive search (ES), i.e., HB with CH achieve higher accuracy than ES (CH) and only deviates 3% from ES (DBI). Hence, the best result of AutoML4Clust deviates only 3% from the best result of ES. Therefore, AutoML4Clust supports novice analysts nearly as good as an exhaustive search, while being more efficient since it does not execute all configurations in $\mathcal{CS}$. Furthermore, we emphasize that AutoML4Clust achieves higher accuracy when using the CH metric. Regarding optimizers, we observe that AutoML4Clust achieves in most cases the best results with the HB optimizer and the CH metric. One possible reason is that BO and BOHB are more effective in higher dimensional configuration spaces. Yet, it achieves higher accuracy than RS since it discards poorly performing configurations early on [3].

4.2.3 Effect of Noisy Data. Throughout our experiments, we observed that noisy data have a significant impact on the baselines. However, noisy data are prevalent in real-world scenarios and should therefore be considered specifically. Figure 4 unveils the results for three different noise ratios $r \in [0; 0.17; 0.5]$. It can be seen that AutoML4Clust is robust against noise, i.e., it achieves AMI values typically over 85% even for $r = 0.5$ and can therefore almost compete with a time-consuming exhaustive search, which achieves an AMI value of 93.9% for DBI. The ES (CH) achieves accurate results for $r = 0$ and $r = 0.17$, while it performs poorly for $r = 0.5$. Hence, it is most affected by noise for $r = 0.5$. We observe a similar behaviour for the AutoML4Clust instantiations with the CH metric. The reason for the bad performance is that the calculation of the CH metric is essentially based on the compactness. Therefore, it is less robust against noise than the DBI metric [14]. However, we note that the results of AutoML4Clust with the CH metric (cf. Section 4.2.2) are still very accurate for $r \in [0; 0.17]$ and are only imprecise for $r = 0.5$, i.e., for highly noisy datasets.

4.3 Runtime Evaluation

Besides an accurate clustering result, the runtime is also crucial for analysts. Figure 5 summarizes the runtime results for all investigated AutoML4Clust instantiations for $l = 60$ and the corresponding baselines.

AutoML4Clust exhibits the highest runtimes with the SIL metric, since this metric has a runtime complexity of $O(n^2)$ [14]. The runtimes of the CH and DBI metrics differ only marginally, while the CH metric has the lower runtime in most cases. Regarding the optimizers, AutoML4Clust achieves faster results with the HB and BOHB optimizers than with the RS and BO optimizers. The reason is that HB and BOHB execute optimizer loops in parallel, while RS and BO execute them sequentially [3].

The results clearly show that AutoML4Clust is orders of magnitude faster than the time-consuming ES. It achieves the fastest results in 57 seconds, while the fastest results for the ES require roughly 6 hours. In comparison to ES (DBI), we even observe speedups of more than 437×. Hence, AutoML4Clust provides an efficient support for novice analysts regarding the CASH problem for clustering analyses.

4.4 Results on Real-World Datasets

In order to assess the practical feasibility of AutoML4Clust, we perform the same experiments as for the synthetic datasets, yet use real-world datasets. We focus on the CH and the DBI metric for these experiments, since the results on the synthetic datasets clearly showed that the SIL metric does not achieve valuable

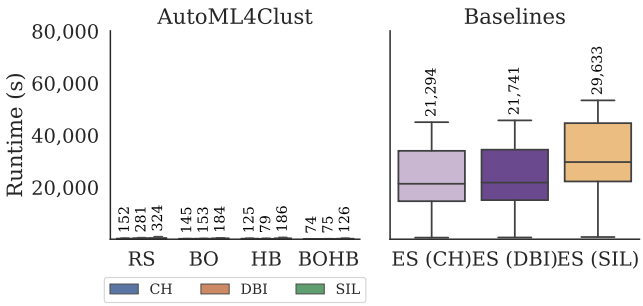


Figure 5: Runtime results of AutoML4Clust with all instantiations in contrast to the respective baselines. Median values are shown at each box plot.

Approach	Optimizer	Metric	AMI (%)	Runtime (s)
AutoML4Clust	RS	CH	**45.4**	781
		DBI	28.5	791
	BO	CH	22.5	879
		DBI	33.6	1,305
	HB	CH	**38.5**	**420**
		DBI	**33.9**	604
	BOHB	CH	22.0	**276**
		DBI	33.7	**598**
Baseline	ES	CH	13.8	76,211
	ES	DBI	33.5	77,196

Table 2: Median results on real-world datasets regarding AMI and the runtimes. We indicate the top-3 results for AMI and runtime in bold.

results and furthermore exhibits high runtimes. Table 2 summarizes the results on the real-world datasets, which are mostly very similar to the results on the synthetic datasets. We indicate the top three results for AMI and runtime in bold.

AutoML4Clust achieves higher accuracy than an exhaustive search, while also significantly outperforming it regarding runtime. The fastest results of AutoML4Clust requires less than 5 minutes, while the ES required with both metrics roughly 21 hours, i.e., AutoML4Clust achieves speedups of up to 276×. Furthermore, AutoML4Clust achieves with HB and CH up to 5% higher AMI values than the ES. The reason that AutoML4Clust can be more accurate than ES is that both optimize the internal metric value of CH or DBI and do not directly address the external metric AMI. Therefore, both approaches return the configuration with the best metric value, but not necessarily with the best accuracy, i.e., AMI value. We argue that HB and CH is a well-performing instantiation of AutoML4Clust, since it is the only instantiation that achieves one of the top-3 results with both, AMI and runtime.

Combining these observations with the results from synthetic datasets, we can state that the instantiation of the HB optimizer and the CH metric achieves in most cases the best results regarding accuracy and runtime.

5 CONCLUSION

In this work, we propose AutoML4Clust, an AutoML approach to support novice analysts efficiently with the combined algorithm selection and hyperparameter optimization (CASH) problem for clustering analyses. To the best of our knowledge, this is the first thoroughly elaborated AutoML approach for clustering analyses. AutoML4Clust remains generic, i.e., it can be instantiated with different optimizers and internal metrics. However, the concrete instantiation is crucial for an efficient exploration of large configuration spaces. Our evaluation reveals that specific instantiations of AutoML4Clust achieve similar or even more accurate results, while tremendously outperforming existing approaches regarding runtime on synthetic and real-world datasets.

Future work will address how clustering ensembles can be exploited to achieve even more valuable clustering results.

REFERENCES

[1] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *Journal of Machine Learning Research* 13, 1 (2012).

[2] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer-Verlag, Berlin, Heidelberg.

[3] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning.*

[4] Daniel G Ferrari and Leandro Nune de Castro. 2015. Clustering algorithm selection by meta-learning systems: A new distance-based problem characterization and ranking combination methods. *Information Sciences* (2015).

[5] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Advances in neural information processing systems.*

[6] Manuel Fritz, Michael Behringer, and Holger Schwarz. 2019. Quality-driven early stopping for explorative cluster analysis for big data. *SICS Software-Intensive Cyber-Physical Systems* (2019).

[7] Manuel Fritz, Michael Behringer, and Holger Schwarz. 2020. LOG-Means: Efficiently Estimating the Number of Clusters in Large Datasets. *Proc. VLDB Endow.* 13, 12 (July 2020).

[8] Manuel Fritz and Holger Schwarz. 2019. Initializing k-Means Efficiently: Benefits for Exploratory Cluster Analysis. In *On the Move to Meaningful Internet Systems: OTM 2019 Conferences.*

[9] Manuel Fritz, Dennis Tschechlov, and Holger Schwarz. 2020. Learning from Past Observations: Meta-Learning for Efficient Clustering Analyses. In *Big Data Analytics and Knowledge Discovery*, Min Song, Il-Yeol Song, Gabriele Kotsis, A Min Tjoa, and Ismail Khalil (Eds.). Springer International Publishing, Cham, 364–379.

[10] Junhao Gan and Yufei Tao. 2015. DBSCAN revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data.*

[11] Anil K Jain. 2010. Data clustering: 50 years beyond K-means. *Pattern recognition letters* 8 (2010).

[12] L. Kaufman and P.J. Rousseeuw. 1987. Clustering by means of Medoids. In *Statistical Data Analysis Based on the L1–Norm and Related Methods.*

[13] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: a novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research* 1 (2017).

[14] Yanchi Liu, Zhongmou Li, Hui Xiong, Xuedong Gao, Junjie Wu, and Sen Wu. 2013. Understanding and enhancement of internal clustering validation measures. *IEEE Transactions on Cybernetics* 3 (6 2013).

[15] J MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability.*

[16] Bruno Almeida Pimentel and André C.P.L.F. de Carvalho. 2019. A new data characterization for selecting clustering algorithms using meta-learning. *Information Sciences* (3 2019).

[17] D. Sculley. 2010. Web-scale k-means clustering. In *Proceedings of the 19th International Conference on World Wide Web, WWW '10.*

[18] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '13.* ACM Press.

[19] Nguyen Xuan Vinh, Julien Epps, and James Bailey. 2010. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *Journal of Machine Learning Research* (2010).

[20] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, S Yu Philip, et al. 2008. Top 10 algorithms in data mining. *Knowledge and information systems* 14, 1 (2008).

Short Paper

Feature-driven Time Series Clustering

Donato Tiano
donato.tiano@univ-lyon1.fr
Lyon 1 University
Lyon, France

Angela Bonifati
angela.bonifati@univ-lyon1.fr
Lyon 1 University
Lyon, France

Raymond Ng
rng@cs.ubc.ca
University of British Columbia
Vancouver, Canada

ABSTRACT

The problem of clustering time series has several applications in real-life contexts, especially in data science and data analytics pipelines. Existing time series clustering algorithms are ineffective for feature-rich real-world time series since they only compute the similarity of time series based on raw data or use a fixed set of features. In this paper, we develop a feature-based semi-supervised clustering framework addressing the above issues for variable-length and heterogeneous time series. Specifically, we rely on a graph encoding of the time series that is obtained by considering a high number of significant extracted features. We then employ community detection and leverage a co-occurrence matrix in order to group together all the best clustering results. Our extensive experimental assessment shows the scalability and robustness of our approach along with its superiority against state of the art clustering algorithms on both real-world healthcare data and UCR benchmark data.

1 INTRODUCTION

The goal of clustering is to organize unlabeled data objects into homogeneous groups while minimizing intra-cluster dissimilarity and maximizing inter-cluster dissimilarity [9]. In this paper, we present FeatTS, a Semi-Supervised Clustering method that leverages features extracted from the raw time series to create clusters that reflect, as much as possible, the original time series. The FeatTS algorithm leverages the concepts of Constrained Clustering, more specifically Clustering by Seeding. The most prominent approach in this category is Seeded kMeans [3], which relies on a small amount of labels of the original dataset in order to create two kinds of links, i.e. Must Link and Cannot Link. Must links are connections between two data points that represent a "constraint of belonging". This means that the data points (or time series at large) should be clustered together. Cannot links do the opposite thus leading to separate data points. Leveraging these two kinds of links, Seeded kMeans allows to discover clusters that respect them.

Our approach differs from existing methods in the literature since it employs the features of the time series, whereas existing methods focus on the similarity of the time series themselves [20]. The novelty of FeatTS consists in automatically selecting the most appropriate statistical features based on the dataset provided as input, the latter characteristic being relevant for data science and data analytics pipelines. In fact, not all the features have the same quality and choosing a subset of high-quality features for each dataset is beneficial for the clustering step. Moreover, the features of time series are interpretable by humans, thus leading to a more transparent and human-centric clustering process. To the best of our knowledge, FeatTS is the first feature-based semi-supervised clustering framework with these characteristics.

In designing our approach, we were inspired by the peculiarities of real-life time series, in particular those found in the time series of patients suffering from end-stage kidney diseases. These time series describe the change over time of the Glomerular Filtration Rate(GFR) signal, estimating how much blood passes through the glomeruli each minute. How GFR changes is crucial for the patient's survival, since rapidly descending values of GFR over time indicate a dangerous condition that might lead to kidney failure and even death. A more stable evolution of GFR over time is less worrisome for the concerned patient, thus guiding an appropriate treatment. The key question to tackle is how to select the subset of features that help medical doctors to discriminate the patients based on the label while performing clustering.

Once the features are selected, FeatTS computes the global relationships between the time series based on their statistical features. We use graph networks to obtain such an encoding. Indeed, FeatTS converts each time series into nodes and creates weighted edges. Each edge represents the distance between the connected nodes of the edge, i.e. the difference between the values of two different time series using the selected feature. FeatTS prunes the graphs based on a threshold and applies a Community Detection algorithm in order to obtain the global relationship among time series. As a final step, FeatTS will holistically merge the results of the communities into a Co-Occurrence matrix, on which clustering (K-Medoid) is applied. Figure 1 depicts at a high level the various steps of our proposed algorithm, while a detailed explanation is provided in the rest of the paper.

As already observed, in the supervised feature selection step, FeatTS allows to select the most appropriate statistical features based on the labels of the time series. The selected features are then used to cluster the entire dataset, including the time series that may have unknown labels. Using Figure 2(a) as a running example on GFR data, our algorithm allows us to obtain the resulted clusters for the four time series reported in Figure 2 (d) even with missing labels for TS_2 and TS_4.

An observant reader may question how this clustering task is different from classification, in which features are selected to build classifiers separating the time series based on the class labels. The key advantage of the clustering task we perform here is that the number of clusters to be formed can be arbitrarily different from the number of classes. For instance, in our kidney failure example, medical researchers may want more clusters to be formed than the two classes of kidney failures or not. Notice that the same separation cannot be achieved with classification [2] as a classifier cannot "sub-divide" the "kidney failure" label. We summarize our main contributions as follows:

(1) We introduce a novel semi-supervised clustering method leveraging the most discriminating features extracted from the time series. We treat the time series similar to each other as communities and we encode the different communities into a co-occurrence matrix, allowing to obtain a unified similarity value for the time series.

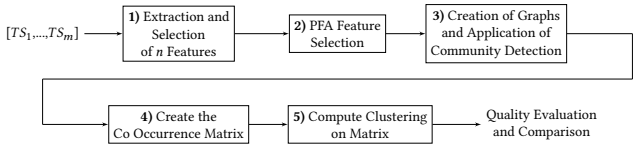


Figure 1: The algorithmic pipeline of FeatTS.

(2) Contrarily to previous work, our method allows to treat at par all the features of a given dataset instead of preselecting a fixed number of them for all datasets or just leveraging the similarity of raw data.
(3) Our method achieves more high-quality results compared with the latest baselines (among which Seeded kMeans [3] and k-Shape [17]) on the literature datasets. In addition, it obtains excellent results in terms of scalability.

The paper is organized as follows. Section 2 discusses the related work. In Sections 3, we describe in detail the steps of our clustering framework. In Section 4, we describe our experimental setup on real-life and on benchmarking data. Section 5 presents the various results of our experimental assessment. Finally, Section 6 concludes our work and discusses future directions.

2 RELATED WORK

Semi-supervised learning is a combination of supervised and unsupervised learning. It uses a small amount of labeled data and a large amount of unlabeled data in order to train a model. As such, it avoids the problem of finding a large amount of labeled data. A subcategory of semi-supervised clustering is Constrained Clustering, that enables the creation of Must Link and Cannot Link, as already explained in Section 1. There exist various methods to create these constraints; among which active learning [19] by relying on the user to provide such constraints. Another method would exploit the labelled data provided on input, as done by Seeded KMeans[3]. The latter uses the labels provided as input to find the constraints among the data and the centroids and then applies the kMeans algorithm to find the clusters.

In the literature, approaches similar to ours exist that perform time series clustering. However, other feature-based approaches [15, 21] only consider a predefined set of features that are limited with real-life time series exhibiting a richer number of features.

Among the unsupervised clustering algorithms [1], kShape[17] can be considered as the state of the art algorithm. It computes the most representative time series in a given cluster and inserts each new time series into one of these clusters based on distance. The problem with this approach is that it can often lead to assigning spurious time series to clusters. As shown in the literature [7], the usage of raw time series can spur high noise levels.

3 A FULL-FLEDGED PIPELINE

The pipeline of FeatTS as illustrated in Figure 1 is a combination of steps that contribute to the quality of the clustering results. We describe these steps in detail in the following.

3.1 Feature Extraction and Selection

The first step of the pipeline is the *feature extraction* step from the time series corresponding to step 1) in Figure 1. We consider feature extraction methods available in the literature and in readily predefined libraries [5]. Formally, given a vector of features $[f_1, f_2, \ldots, f_n]$ extracted, we construct a table containing the value of each feature, thus having as columns the features and having the time series $[TS_1, TS_2, \ldots, TS_m]$ as rows. As a simple example, in Figure 2(a), we show an instance of the table for 4 time series and 6 features. Moreover, each time series is displayed with its corresponding label.

The features shown in Figure 2(a) represent only a tiny subset of the set of features that can be extrapolated from the time series. Indeed, the tsfresh[5] library allows us to extract a significantly higher number of features. Therefore, *feature selection* becomes pivotal in our setting, since not all the features have the same relevance for the subsequent clustering steps. In particular, we compute the relevance of the extracted features by solely using the feature values corresponding to the class label of the time series (e.g. in Figure 2(a) the class 'Kidney Failure' or ' No Kidney Failure').

The Benjamini-Yekutieli is a supervised procedure [4] that allows us to identify the relevance of the features, based on the label associated with the time series. It computes the p-value of each feature provided as input based on their relevance. The p-value is an important metric that allows us to quantify the significance of each feature. The output of Benjamini-Yekutieli procedure is a list of features ranked by their p-values. Among these features, usually only a subset of them have an acceptable relevance. From our empirical study, it has been evinced that the top-20 features in order of relevance are sufficient to obtain high-quality clustering.

Usually, one of the main problems when computing the p-value is that the redundancy of the obtained features. It is desirable to find a duplicate-free combination of the features that is still quality preserving and small in number. Therefore, once we select the 20 features from the list produced by Benjamini-Yekutieli, we need an algorithm that allows us to find a minimum number of features that is representative of the other features not included in the analysis.

To do so, we apply a technique called Principal Feature Analysis (PFA) [12]. PFA is a variation of Principal Component Analysis (PCA). The key difference is that PFA preserves the original values of the features and thus the distance between them. Thus, we can leverage the concept of explained variance, representing the ratio between the variance of one single feature and the sum of variances of all individual features. We fix a value t of the explained variance, which is in our experiments equal to 0.9. That is, out of the 20 features selected in the Benjamini-Yekutieli procedure, we choose the minimum number of features for which the sum of their variance covers the 90% of the information produced by the rest of the features. This value is the best result produced empirically with various values of the threshold t. In our example, among all the features presented in Figure 2(a), we have selected only *quantile*, *trend_stderr* and *trend_rvalue*, as shown in Figure 2(b).

3.2 Graph Rendering and Community Detection

We convert the time series and their relationships into edge-weighted graphs. The encoding of time series into edge-weighted graphs allows us to represent our clustering problem in another dimension space without loss of information. This operation is crucial in order to be able to capture the global relationships among the raw time series samples.

Suppose we have a feature F_i (as selected by PFA in the previous step) and a set of n time series $\{TS_1, .., TS_n\}$. Let TS_i be a node v_i in the set of vertices V of a graph G. Let E be the set of

a

Time Series	*mean*	*trend_stderr*	*variance*	*peaks*	*quantile*	*trend_rvalue*	Length	Label
TS_1	51.3	3.51	788.56	8	57	-0.94	89	No Kidney Failure
TS_2	40.6	4	128.9	5	43	-0.55	206	~~No Kidney Failure~~
TS_3	74.3	17	296.8	10	106	0.01	159	Kidney Failure
TS_4	95.8	9.4	783.3	10	85	0.43	139	~~Kidney Failure~~

b

quantile
trend_stderr
trend_rvalue

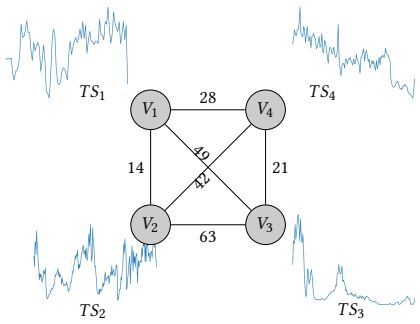
c

Dataset	TS_1	TS_2	TS_3	TS_4
TS_1	1	$\frac{1+0.5}{0.66+1+0.5}$	$\frac{0.5}{0.66+1+0.5}$	$\frac{0.5}{0.66+1+0.5}$
TS_2	$\frac{1+0.5}{0.66+1+0.5}$	1	$\frac{0.5}{0.66+1+0.5}$	$\frac{0.5}{0.66+1+0.5}$
TS_3	$\frac{0.5}{0.66+1+0.5}$	$\frac{0.5}{0.66+1+0.5}$	1	$\frac{0.66+1+0.5}{0.66+1+0.5}$
TS_4	$\frac{0.5}{0.66+1+0.5}$	$\frac{0.5}{0.66+1+0.5}$	$\frac{0.66+1+0.5}{0.66+1+0.5}$	1

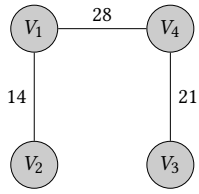
d

Dataset	TS_1	TS_2	TS_3	TS_4
TS_1	1	0.69	0.23	0.23
TS_2	0.69	1	0.23	0.23
TS_3	0.23	0.23	1	1
TS_4	0.23	0.23	1	1

Figure 2: A running example on real-world healthcare data.



(a) Edge-weighted graph with distances as weights.



V_1	V_2	**14**
V_3	V_4	**21**
V_1	V_4	**28**
—	—	—
V_2	V_3	42
V_1	V_3	49
V_2	V_4	63

(b) The graph for a single feature after application of the threshold

Figure 3: Encoding from time series to graph.

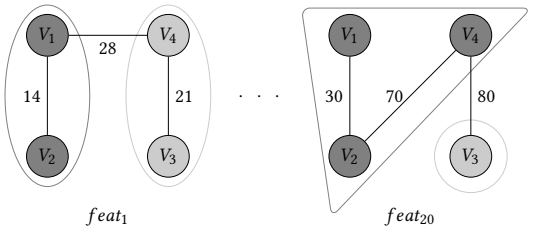


Figure 4: Application of Community Detection algorithm for each feature.

edges of graph G, where each edge e_i connects two nodes in G representing two distinct time series. Let $w : E \rightarrow \mathbb{R}$ be an edge-based weight function. Each edge e_i is thus assigned a weight $w(e_i)$ representing the distance between the connected nodes of the edge, i.e. the difference between the values of two different time series using the feature F_i. In order to capture similarity, we only retain in G the edges whose weight is less than a given threshold distance th.

Example 3.1. As an example, let us consider the four time series as in Figure 2(a), each of which has the values of quantile; we will compute all the distances between these values. Figure 3a shows the graph encoding of these time series where the weights on the edges represent the distance between the time series, based on quantile.

One immediate question is the choice of the threshold th. Given n nodes in G corresponding to the n time series, there are $\frac{N*(N-1)}{2}$ distances between all pairs of nodes. To capture similarity, we use a simple heuristic of a percentage x that represents the proportion of the smallest distances to be kept. The threshold th is thus selected based on this x percentage.

Example 3.2. For instance, for the graph in Figure 3a, the array in Figure 3b contains the distances between the vertices in ascending order. Suppose that the user specifies as percentage 50% of the vector. This implies that the distance boundary will be 28 and the distances higher than 28 will be discarded (i.e. the corresponding edges in the graph will be ignored). Once we have chosen the boundary distance, we can create the corresponding graph as depicted in Figure 3b.

Notice that a higher threshold would consider lower significance edges and thus weaker similarities between the times series. On the other hand a lower threshold may cut important edges. In our empirical evaluation, we used a threshold determined by a user-specified percentage of 80%, which works well in practical scenarios as we will see in the remainder of the paper. The chosen threshold will be used for all features selected by PFA and thus for all graphs created.

Note that each graph is created based on one selected feature from PFA in the previous step. Thus, if PFA selects k features, there will be k graphs, each corresponding to one notion of similarity between a pair of time series. The intention is to combine these different notions of similarity in time series clustering, by leveraging the structures of connectivity in the various graphs.

To this purpose, we apply a community detection (CD) algorithm in order to search for groups of densely connected vertices forming communities. Among the different tested algorithms, we have opted for the Greedy Modularity Algorithm [16] in the NetworkX library [8]. This algorithm turns out to strike a balance between speed and robustness and does not require any additional input parameter other than the graphs.

In Figure 4, we show an example of clustering obtained by applying this algorithm to a family of graphs. We can notice that the clustering varies from one graph to another graph. A natural question is how we can unify the different clusters in order to obtain understandable results.

3.3 Creation of the Co-Occurrence Matrix

The underlying intuition is that if two time series are similar, they will be similar for the majority of their discriminating features.

We employ a *co-occurrence matrix* [14] to put this in practice. The matrix consists of recording for each pair of time series how many times they are grouped within the same community. Intuitively, the more times they are placed within the same community, the more similar the time series are.

Co-Occurrence Matrices without weights. Assuming we have M time series and L features, we know that, once applied the CD algorithm on the L graphs, we will obtain the following result:

$$\begin{gathered} Feature_1 = \{(TS_1, TS_3, ..., TS_s), ..., (TS_2, TS_4, ..., TS_p)\} \\ Feature_2 = \{(TS_2, TS_3, ..., TS_t), ..., (TS_1, TS_4, ..., TS_i)\} \\ ... \\ Feature_n = \{(TS_2, TS_1, ..., TS_m), ..., (TS_3, TS_4, ..., TS_q)\} \end{gathered}$$

where for each feature $Feature_i$ selected by the PFA, we obtain different communities $(TS_1, ..., TS_i)$ composed by the time series. We can now create a matrix in which the rows and columns contain all the time series of the dataset. Each cell x_{ij} in the matrix corresponds to the similarity between time series TS_i (in row i of the matrix) and TS_j (in column j of the matrix).

Next we convert the counts in the Co-Occurrence matrix into a similarity metric for the eventual clustering. We consider the number of times that a pair of time series is present in all possible communities where at least one of the two time series belongs. That is, given the time series TS_i, TS_j, the communities C and the set of all the time series M, the similarity between TS_i and TS_j will be as follows.

$$\forall\, TS_i, TS_j \ \in M,\ \forall c \in C\ x_{ij} = \frac{\left|\{\, c \in C \mid TS_i \in c \quad \& \quad TS_j \in c \,\}\right|}{|\{c \in C |\ TS_i \in c\}|} \tag{1}$$

That is, the number of times that the two time series TS_i and TS_j fall within the same community divided by the number of times that TS_i is found within any community.

Notice that (1) is completely symmetrical. Indeed, the communities found for each feature are considered as hard clustered, i.e. a time series TS_i cannot be part of two communities of the same feature and must necessarily belong to one community. Thus, if TS_i and TS_j are within the same community of a specific feature, neither of them can be part of other communities of the same feature. Therefore, the value x_{ij}, given by the number of times TS_i and TS_j are in the same community, will be equal to x_{ji}, because TS_j and TS_i must also be in the same community.

Co-Occurrence Matrices with Weights. The application of the CD algorithm and its processing with co-occurrence matrices without weights might incur the problem of community fragmentation. More precisely, the CD algorithm might lead to the formation of a high number of communities each of which contains a few time series. This is due to the fact that some features are often not discriminatory enough for that dataset.

To overcome this problem, we assign an approximate weight to each feature, based on the number of communities that the CD algorithm derives from the graph. Again, to correctly determine the weights, we require the input of the user on the expected number of clusters.

Let w_i be a weighting function defined on each feature F_i as follows:

$$\begin{cases} w_i = \frac{C}{O_i},\ if\ O_i > C \\ w_i = \frac{O_i}{C},\ if\ C > O_i \\ w_i = 1,\ otherwise \end{cases} \tag{2}$$

Dataset	TS_1	TS_2	TS_3	TS_4
TS_1	0	0.64	1.36	1.36
TS_2	0.64	0	1.36	1.36
TS_3	1.36	1.36	0	0
TS_4	1.36	1.36	0	0

Table 1: Co-Occurrence Matrix with weights.

where C is the number of clusters expected by the user and O_i is the number of communities extracted by means of the CD algorithm. Hence, the weights will be higher if the number of obtained communities O is equal or sufficiently close to C and lower otherwise.

The weights will be propagated to the similarity matrix, which will now reflect the importance of each feature from a user viewpoint. Not surprisingly, instead of simply counting the number of times that the time series TS_i and TS_j co-occur in the same community, we now sum their weights and divide by the sum of the weights of all time series, as also shown in the following.

Example 3.3. As shown in Figure 2(b), the PFA feature selection has chosen three features, namely [$trend\_stderr$, $quantile$, $trend\_rvalue$] *. After applying the CD algorithm, we obtained the following communities (per feature) among the* 4 *Time Series in Figure 2(a):*

$$\begin{gathered} quantile = \{(TS_1, TS_2), (TS_3, TS_4)\} \\ trend\_stderr = \{(TS_1), (TS_2), (TS_3, TS_4)\} \\ trend\_rvalue = \{(TS_1, TS_2, TS_3, TS_4)\} \end{gathered}$$

Assume now that the user specified an expected number of clusters equal to 2. $Trend\_stderr$ *and* $trend\_rvalue$ *do not satisfy the number of clusters expected by the user. Therefore,* $trend\_stderr$ *will have a weight of* $\frac{2}{3}$ *(*0.66*), while* $trend\_rvalue$ *will have a weight of* $\frac{1}{2}$ *(*0.5*), and* $quantile$ *we will have* 1 *as weight. We report the intermediate computation of the co-occurrence matrix with weights for this example in the Table in Figure 2(c) and the final result in the Table in Figure 2(d).*

3.4 Clustering the Co-Occurrence Matrix

The co-occurrence matrix obtained in the previous step allows us to quantify the similarity between two time series. In order to be prepared for the creation of the time series clusters, we need one more intermediate step, i.e. to compute the distances between the rows of the Co-Occurrence Matrix. We employ a standard Euclidean distance to perform the row comparison.

As an example, applying the Euclidean distance between the rows of the table in Figure 2(d), we obtain Table 1. For instance, the value of the cell $C_{3,4}$ of the Table 1 is 0 because the row 3 and 4 of the Table in Figure2(d) are equal. As a consequence, the time series 3 and 4 are always together in each cluster discovered by the CD algorithm.

Finally, we apply the standard K-Medoid algorithm [10] on the distances computed above. K-Medoid allows us to extract the time series that have the smallest distance among them.

To complete our running example, applying the K-Medoid algorithm to Table 1 requiring 2 clusters as the input parameter, we obtain two clusters $Cl_1 = \{TS_1, TS_2\}$ and $Cl_2 = \{TS_3, TS_4\}$, as shown in the Table in Figure 2(d).

The time complexity of the FeatTS is summarized in Lemma 3.4. The proof of the Lemma is available in the online repository[1].

Lemma 3.4. *Let D a dataset composed by m time series and let L the number of features chosen by PFA among the N features*

[1] https://github.com/DonaTProject/FeatTS

extracted and k the number of requested clusters. A dataset D is evaluated by FeatTS in time $O(L(m^2) + m^2 + k(m-k)^2 + n \cdot t_f)$ *and in space* $O(n + L(m + E) + m^2)$.

4 EXPERIMENTAL SETUP AND A CLINICAL CASE STUDY

We use real-life time series courtesy of the Personalized Medicine Department at the European Hospital George Pompidou in Paris. These time series contain signals from patients suffering from kidney diseases. As a background, the human kidney has a lot of functions including maintenance of acid-base balance. Proper function of the kidney requires that it receives and adequately filters blood. This is performed at the microscopic level by many hundreds of thousands of filtration units called renal corpuscles, each of which is composed of a glomerulus. A global assessment of renal function is often ascertained by estimating the rate of filtration, called the glomerular filtration rate (GFR). GFR estimates how much blood passes through the glomeruli each minute. Kidney failure occurs when GFR is under $90mL/min/1.73m^2$, whereas when it drops to $15mL/min/1.73m^2$, it means that the patient needs dialysis or a transplant. Thus, it is very important to understand when a patient needs medical treatment before the GFR reaches its lowest possible value. Moreover, since dialysis is an invasive operation, it is important to understand if a sudden drop in the GFR occurs. In this case, medical doctors might recommend urgent surgery or to resort to dialysis depending on the GFR values over time.

We ran experiments on two variants of this dataset. The first variant named $Kidney3Yr$ contains 222 patients (one time series per patient) and spans 1 to 3 years with a variable length between 90 and 230 data points in the time series. The second variant called $Kidney5Yr$ is composed of 278 patients spanning 5 years with time series having roughly 100 data points. In both cases, we ran our experiments using only the 20% of the labeled time series in order to compute the set of features necessary to run the clustering algorithm and to emulate the real-world scenario where not all the labels of the data points are available. The features thus being ordered based on their relevance have been employed to cluster the entire unlabeled dataset into those with GFR signals concerning high-risk patients and those containing GFR for patients with lower risk.

UCR datasets. We also used 64 benchmark datasets belonging to the UCR collection[6], including both real-life and synthetic datasets. The entire list of datasets used for benchmark is available online [2]. For consistency, we also used only 20% of the labeled time series during feature extraction during the clustering step in all experiments.

Implementation and reproducibility. Our code base is available online [2] with more details about the reproducibility.

5 EXPERIMENTAL RESULTS

For each dataset, we consider 20 as the upper bound of the number of features we consider in the analysis. A higher number of features are supported by our method but not indispensable to obtain better accuracy. Moreover, a higher number of features deteriorates the performance. Furthermore, we chose 80% as the threshold value of the percentage of features selected by the user. We used the AMI [18] metric, which is a well-established measurement of the quality of clustering. We adopt the same metric for our comparisons as well.

[2]https://github.com/DonaTProject/FeatTS

Dataset	FeatTS	kShape	SeededKMeans
Adiac	0,31	0,39	**0,52**
MoteStrain	**0,48**	0,01	0,02
TwoLeadECG	**0,88**	0,10	0,07
ECG200	**0,34**	0,11	0,06
Computers	0,09	**0,06**	0,01
Coffee	**1**	0,35	0,88
GunPoint	**0,52**	0	0
Arrowhead	**0,29**	0,26	0,27
ItalyPowerDemand	**0,54**	0,39	0
Meat	0,4	0,64	**0,75**
OliveOil	0,27	0,52	**0,53**
Trace	**0,74**	0,52	0,69
Wine	**0,12**	0	0,01
Worms	**0,16**	0,06	0,12
ShapesAll	0,08	0,62	0,45

Table 2: Results showing the values of AMI for UCR datasets

We consider two main baselines, the first being the state-of-the-art algorithm for time series clustering, i.e. KShape[17], and the second being the state-of-the-art algorithm for Semi Supervised time series clustering i.e. Seeded kMeans [3], sharing the same category of our approach (as detailed in Section 2). KShape[17] could not be used on the real-world GFR time series as it cannot process variable-length time series. Hence, we limit the comparison with KShape to the UCR datasets.

Other baselines of semi-supervised clustering algorithms such as SSSL[20] (Self Labeling Algorithm) and SUCCESS[13] (Cluster then Label Algorithm) could not be used in our study due to the lack of available source code.

All experiments have been executed on a Server running Linux with 64GB of RAM, Intel Xeon CPU Skylake,IBRS @ 2.6GHz.

5.1 UCR Dataset

The results in Table 2 are an excerpt of the entire results obtained by our algorithm and its baselines for various UCR datasets. It can be observed that FeatTS obtains the best results among all.

Indeed, out of 64 datasets used for the comparison, FeatTS performed better on 37 datasets. In addition, kShape performed well only on 15 datasets (out of 64) and Seeded kMeans outperformed the others in only 12 datasets (out of 64).

5.2 Kidney Case Study

As shown in Table 3, the results obtained by FeatTS are significantly more accurate than Seeded kMeans on the clinical case study. For the patients under medical supervision for 5 years, as shown in Table 3, we have obtained results following a similar trend.

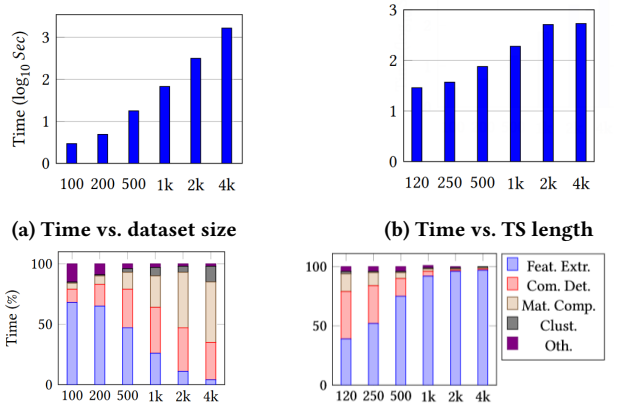
Dataset	FeatTS	SeededKMeans
Kidney3Yr	**0.56**	0.44
Kidney5Yr	**0.58**	0.48

Table 3: Results on Kidney 3Yr and 5Yr Datasets

5.3 Scalability

We have assessed the scalability of our method by increasing both the number and length of time series in a dataset. In this experiment, we have used synthetic time series generated with GRATIS [11]. This tool allows a controlled generation of time series by using diverse characteristics as spectral entropy, trend, seasonality, stability, etc.

In our case, in the synthetic generation we have opted for spectral entropy and trend as the underlying characteristics since they reflect the real-life time series we have used in the rest of our experimental assessment. The spectral entropy allows to



(a) Time vs. dataset size **(b) Time vs. TS length**

(c) % Time of each component vs. dataset size. **(d) % Time of each component vs TS length.**

Figure 5: Scalability Results.

measure the "forecastability" of a time series. It has a range of values between 0 to 1 and a low value of entropy indicates a high signal-to-noise ratio, while large values occur when a time series is difficult to forecast. For this characteristic, we have fixed a value of spectral entropy equal to 0.6. Conversely, the trend allows to represent the occurrence of low-frequency variations in the time series and as such it is opposite to seasonality. It has a range of values between 0 to 1 and we have chosen a value of 0.9 for this experiment.

In the first experiment, we increase the number of time series for each tested dataset while the the length of the time series is fixed and equal to 60. Figure 5a shows the results obtained on datasets consisting of 100, 200, 500, 1000, 2000, 4000 time series, respectively. The results show the scalability of the method in terms of time performance, while an important increase can be observed when shifting to more than 2000 time series. The times in Figure 5a are in logarithmic scale for clarity of exposition.

In order to better understand the results, we have studied the percentage of time due to each component of our pipeline as shown in Figure 5c. Upon increasing the size of the dataset, the component that is computationally more demanding is the creation of the co-occurrence matrix. Obviously, since the Co-Occurrence Matrix depends on the number of time series, the time required for its creation increases as the size increases.

In the second experiment, we have increased the length of the time series while fixing to 500 the number of time series belonging to each dataset. Figure 5b shows the results obtained increasing the length of the time series between 120 and 4000. The figure shows the scalability of the approach for time series under 2000 and a sudden increase of the time beyond this value. Also in this case, the times in Figure 5b are in logarithmic scale for better clarity. The time breakdown in Figure 5d shows that the more expensive step of the pipeline for this experiment is the feature extraction step.

6 CONCLUSION AND FUTURE WORK

Our work on clustering of time series shows that there is no one-size-fits-all solution regarding the set of features to use. In fact, we leveraged the features drawn from the data itself rather than taking a predefined set of features for all the datasets. Our flexible graph encoding allows us to process the most significant features in parallel and the further steps of our method allow us to combine the results.

This work could be improved by rendering the entire pipeline unsupervised instead of the current semi-supervised approach. This requires non-trivial extensions in order to be able to cluster the time series without loss of performance. Another improvement would be to dynamically choose the threshold for graph creation based on the processed features. Finally, the weights of the community detection algorithm could be combined with relevance degrees of the features.

7 ACKNOWLEDGEMENTS

Research funded by ANR (grant nr. 18-CE23-0002 QualiHealth). We thank B. Rance and A. Rogier for kindly providing us with the GFR data.

REFERENCES

[1] Saeed Aghabozorgi, Ali Seyed Shirkhorshidi, and Teh Ying Wah. 2015. Time-series clustering–a decade review. *Information Systems* 53 (2015), 16–38.
[2] Anthony J. Bagnall, Jason Lines, Aaron Bostrom, James Large, and Eamonn J. Keogh. 2017. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery* (2017).
[3] Sugato Basu, Arindam Banerjee, and Raymond Mooney. 2002. Semi-supervised clustering by seeding. In *In Proceedings of ICML*. Citeseer.
[4] Yoav Benjamini and Daniel Yekutieli. 2001. The control of the false discovery rate in multiple testing under dependency. *Annals of Statistics* (2001).
[5] Maximilian Christ, Nils Braun, Julius Neuffer, and Andreas W. Kempa-Liehr. 2018. Time Series FeatuRe Extraction on basis of Scalable Hypothesis tests (tsfresh – A Python package). *Neurocomputing* 307 (2018).
[6] Hoang Anh Dau, Anthony Bagnall, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, and Eamonn Keogh. 2018. The UCR Time Series Archive. arXiv:cs.LG/1810.07758
[7] Levent Ertöz, Michael Steinbach, and Vipin Kumar. 2003. Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data. In *Proceedings of the 2003 SIAM international conference on data mining*.
[8] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11 – 15.
[9] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition.* Springer. https://doi.org/10.1007/978-0-387-84858-7
[10] Anil K. Jain and Richard C. Dubes. 1988. *Algorithms for Clustering Data.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
[11] Yanfei Kang, Rob J. Hyndman, and Feng Li. 2020. GRATIS: GeneRAting TIme Series with diverse and controllable characteristics. *Statistical Analysis and Data Mining: The ASA Data Science Journal* (2020).
[12] Yijuan Lu, Ira Cohen, Xiang Sean Zhou, and Qi Tian. 2007. Feature selection using principal feature analysis. In *Proceedings of the 15th ACM international conference on Multimedia*. 301–304.
[13] Kristóf Marussy and Krisztian Buza. 2013. Success: a new approach for semi-supervised classification of time-series. In *International Conference on Artificial Intelligence and Soft Computing*.
[14] Loris Nanni, Sheryl Brahnam, Stefano Ghidoni, Emanuele Menegatti, and Tonya Barrier. 2013. Different approaches for extracting information from the co-occurrence matrix. *PloS one* 8, 12 (2013), e83554.
[15] Alex Nanopoulos, Rob Alcock, and Yannis Manolopoulos. 2001. Feature-based classification of time-series data. *International Journal of Computer Research* (2001).
[16] Mark Newman. 2010. *Networks: An Introduction.* Oxford University Press, Oxford. 784 pages.
[17] John Paparrizos and Luis Gravano. 2016. K-Shape: Efficient and Accurate Clustering of Time Series. *SIGMOD Record.* (2016).
[18] Simone Romano, Nguyen Xuan Vinh, James Bailey, and Karin Verspoor. 2016. Adjusting for chance clustering comparison measures. *The Journal of Machine Learning Research* 17, 1 (2016), 4635–4666.
[19] Toon Van Craenendonck, Wannes Meert, Sebastijan Dumančić, and Hendrik Blockeel. 2018. Cobras ts: A new approach to semi-supervised clustering of time series. In *International Conference on Discovery Science*. Springer, 179–193.
[20] Haishuai Wang, Qin Zhang, Jia Wu, Shirui Pan, and Yixin Chen. 2019. Time series feature learning with labeled and unlabeled data. *Pattern Recognition* (2019).
[21] Xiaozhe Wang, Kate Smith, and Rob Hyndman. 2006. Characteristic-based clustering for time series data. *Data mining and knowledge Discovery* 13, 3 (2006), 335–364.

Short Paper

Indexed Log File: Towards Main Memory Database Instant Recovery

Arlino Magalhaes
Federal University of Ceara
Federal University of Piaui
Fortaleza, Brazil
arlino@ufpi.edu.br

Jose Maria Monteiro
Federal University of Ceara
Fortaleza, Brazil
monteiro@dc.ufc.br

Angelo Brayner
Federal University of Ceara
Fortaleza, Brazil
brayner@dc.ufc.br

Gustavo Moraes
Federal University of Ceara
Fortaleza, Brazil
gustavomoraes94@gmail.com

ABSTRACT

Main Memory Database Systems (MMDBSs) may significantly increment IOPS (Input/Output Operations per Second) rates by avoiding access to secondary memory. This occurs because they maintain the database in Random Access Memory (RAM). Similar to traditional Disk-Based Database Systems (DBSs), MMDBSs are expected to trigger recovery activities after system failures to restore the database to its last consistent state before the failure. Nonetheless, MMDBSs executes the recovery process in an offline way, thus the database becomes available for new transactions only after the full recovery process has been performed. Systems can keep database replicas for high availability. However, replication is not immune to some failure sources that can cause multiple and shared malfunctions. Therefore, software techniques are required to prevent failures and repair crashed systems as soon as possible. This work proposes a novel MMDBS instant recovery process which makes MMDBS able to schedule new transactions simultaneously with the recovery activities. In order to validate this new approach, simulations with a prototype implemented on Redis have been conducted over Memtier benchmark. The achieved results evidence the suitability of the proposed recovery mechanism.

1 INTRODUCTION

Main Memory Databases (MMDBs), or In-Memory Databases (IMDBs), can provide very high throughput rates given that the primary data are located in memory. In that manner, MMDB reduces secondary memory I/O bottleneck, and can consequently speed up data access. Moreover, the development of new memory technologies has provided a larger storage capacity with lower costs. The fact that the database resides in volatile storage influences the design approaches adopted by MMDBs, such as query processing, concurrency control, recovery after crashes, data storage, and indexing. For this reason, these systems are designed to optimize access to main memory instead of secondary memory, as with traditional disk-resident systems [7, 21, 24].

MMDBs provide very high IOPS given that the primary database is handled in volatile storage. However, the database residing in a volatile memory makes these systems much more sensitive to system failures than conventional disk-resident database systems. The recovery mechanism is responsible for restoring the database to the most recent consistent state before a system failure has occurred. In this way, after a system crash, the recovery manager loads the last valid checkpoint (a prior database backup copy) and then starts to execute all actions recorded in the log file forward from the checkpoint record [10, 12, 13].

Accordingly, the recovery process for most MMDBs is performed offline, meaning that the database and its applications only become available for new transactions after the full recovery process is completed. One may claim that systems can keep database replicas for high availability. In fact, with the advent of high-availability infrastructure, recovery speed has become secondary in importance to runtime performance for most MMDBs [7, 12, 22]. Nevertheless, replication is not immune to human errors and unpredictable defects in software and firmware that are a source of failures and can cause multiple and shared problems [18, 21].

In this sense, this paper proposes an instant recovery approach for OLTP MMDBs. Said approach allows MMDBs to schedule new transactions immediately after the failure during the recovery process, giving the impression that the system was instantly restored. The main idea of instant recovery is to organize the log file in a way that enables efficient on-demand and incremental recovery of individual database tuples.

It is important to note that the existing MMDBs recovery strategy has two deficiencies that make instant recovery impossible. First, the recovery process uses a sequential log file. The recovery in the sequential log is not incremental and requires full recovery before any tuple can be accessed. This scenario does not allow the system to execute an on-demand transaction during recovery, which means that new transactions can only start executing after the recovery process has finished. The second problem is the random access pattern in the sequential log for restoring tuples individually. The sequential log has efficient record writes, but it has inefficient reads for individual log records. A full log scan must be done to restore a given tuple individually [12, 18].

The instant recovery technique presented in this work builds the log file as an index structure. This log organization enables an efficient restoration of a tuple. A single fetch on the indexed log can restore one tuple. Thus, the system can use the indexed log to recover a database by restoring tuple by tuple incrementally. This technique naturally supports database availability because a new transaction can access a tuple immediately after the tuple is restored, i.e., transactions do not have to wait for a full recovery to access restored tuples. We have empirically evaluated the proposed instant recovery approach in order to show its efficiency

and suitability to be implemented in MMDBs. The workload used for the experiments belongs to Memetier benchmark.

The remainder of this paper is organized as follows. Session 2 provides an overview of MMDB recovery. Section 3 discusses related work. Section 4 presents the proposed approach for database instant recovery. Section 5 discusses the results of empirical experiments. Finally, Section 6 concludes this paper.

2 BACKGROUND

Most main memory database systems implement logical logging technique which records higher-level database operations, such as inserting a record in a table. MMDBSs do not record Before images of modified tuples, i.e., they produce only Redo log records to reduce the amount of data written to secondary storage. The commit processing uses group commit, i.e., it tries to group multiple log records into one large I/O. SSD is the log device of choice for almost all systems in order to increase the I/O performance [12, 22, 26].

The recovery component of most MMDBs asynchronously produces a consistent checkpoint, commonly called snapshot. Snapshot is equivalent to a materialized database state in an instant of time by means of a Copy-on-Update method (COU, for short) [4, 6]. At the checkpoint beginning, MMDBS enters into a COU mode. Thereafter, the recovery component starts to scan all tuples in database tables at system run-time. Three bits are used to identify if a row was inserted, deleted, or updated after the checkpoint generation has begun. Inserted tuples are disregarded. Before an update or delete, the original content of a tuple is copied to a shadow table. The shadow version is removed after it is scanned. A background process serializes the snapshot to secondary memory until the checkpoint ends [2, 12].

Whenever a system crash occurs in an MMDB, the primary copy of the database is lost. In this case, MMDBs recover the database by loading the last valid checkpoint. Thereafter, the recovery component starts to execute the actions recorded in the log file forward from the checkpoint record. The recovery component activities are briefly illustrated in Figure 1. All actions of committed transactions are flushed to the log file on secondary memory by the Logger component. Periodically, the Checkpoint component produces a snapshot on secondary storage. After a failure, the Restorer component loads the snapshot into memory and then replay the log file. After the recovery process has finished, the database is available for new transactions [7, 22].

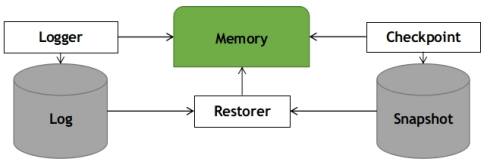


Figure 1: MMDBS recovery component architecture.

3 RELATED WORK

Hekaton [5], VoltDB [20], HyPer [9], SAP HANA [8] and SiloR [26] are examples of modern MMDBSs that perform the recovery activities discussed in Section 2. Nevertheless, those systems do not execute new transactions until the full recovery is completed.

PACMAN [22] and Adaptive Logging [23] utilize a dependency graph between transactions performed to identify opportunities for database recovery in parallel. After a failure, they use a dependency graph to generate a scheduled execution to replay the log records. The schedule allows transactions to be executed in parallel, following the constraints of the dependency graph. Those systems must wait for the full database recovery to service new transactions.

The Log-Structured Merge tree (LSM-tree) [15] provides low-cost indexing for a file that has a high rate of record insertions and deletions. However, the LSM-tree access method uses a buffer to avoid multiple I/Os in secondary memory for frequently referenced pages. This approach is not suitable for writing log records since they require immediate and atomic persistence during commit processing.

FineLine [17–19] presents an instant database restoration technique. This technique uses a partitioned index in the log to write records efficiently. The partition index may search for multiple partitions to retrieve a page. After a crash, the recovery process loads pages incrementally from a backup device. While a set of pages is loaded, the records in the log file that are related to that set are probed. This approach also can recover pages on-demand for transactions. New transactions can perform as soon as necessary pages are restored.

4 A NOVEL INSTANT RECOVERY MECHANISM

In this section, firstly the architecture of the proposed instant recovery mechanism is described. Thereafter, the proposed log data structure is discussed. Finally, the recovery algorithm based on the proposed indexed log structure is detailed.

4.1 The Architecture

Figure 2 proposes an architecture to implement our MMDB instant recovery approach, which uses an indexed log structure. This section provides an overview of the main components that comprise the architecture and their interactions.

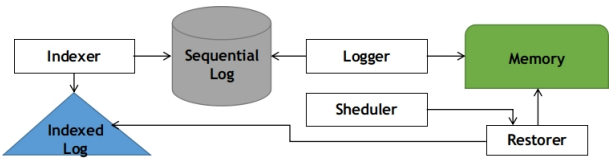


Figure 2: Architecture for in-memory database instant recovery using an indexed log.

Below, we present a brief description of the main components of the architecture discussed in the following subsections.

- **Logger**: writes transaction update actions in a log file on secondary memory. The records are flushed sequentially in an append-only file whose writes must be synchronized to transaction commit.
- **Indexer**: indexes records from sequential log to indexed log. The records are indexed in a B-tree asynchronously to transaction commit.
- **Restorer**: restores tuples from a failed database by replaying records from indexed log. Tuples can be restored incrementally and on-demand.
- **Scheduler**: during the recovery process, requests the Restorer component for tuples (that have not yet been restored into memory) requested by new transactions.

4.2 The Logging Strategy

The proposed approach for MMDB instant recovery uses two logs: a sequential log (Figure 3 (a)), and an indexed log (Figure 3 (b)). Each record in the sequential log represents an update performed on a tuple by a transaction. During transaction processing, transaction update records are appended to the sequential log file by the Logger component. Each transaction generates Redo records that are kept in a thread-local. During the commitment, all log records generated by a transaction are appended atomically on the sequential log. This scheme ensures log consistency to recover the database.

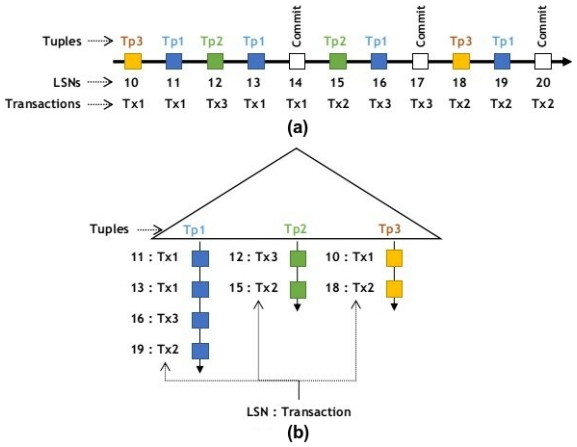


Figure 3: Sequential log (a), and indexed log (b).

The proposed recovery scheme requires efficient log reading to fetch the records to redo a given tuple during recovery. For this reason, the logging strategy implements an indexed log. The index structure is a B-tree in which each node contains a tuple ID and the update records generated by transaction updates in the tuple. Only one probe on B-tree can retrieve all the necessary records to restore a single tuple. The Indexer component is responsible for indexing records from the sequential log to the indexed log. The indexing is asynchronous to transaction commit, i.e., a transaction does not need to wait for the log indexing to confirm its writing in the data. Records can be removed from the sequential log after they are indexed in the B-tree. However, the sequential log is maintained to ensure consistent database recovery in the event of index corruption. In this case, the system must build a new indexed log from the sequential log. This process will delay the start of recovery.

The primary purpose of instant recovery is to restore the database efficiently, without degrading the transaction throughout provided by the system. The indexed log requires random writes, while a sequential log has a sequential write pattern. Writing records to a sequential log file is potentially faster than doing so to an indexed log file. For this reason, in our approach, log records are written to the sequential file and, periodically, flushed to the indexed log file. The indexing process occurs asynchronously to the transaction commit operation so as not to degrade the transaction processing. It is important to highlight that restoring an individual tuple by indexed log requires only one fetch on B-Tree, while restoring a single tuple by sequential log requires a full scan of the log file. Therefore, our recovery technique only uses the indexed log to recover the database after a failure.

FineLine [17–19] uses an instant recovery technique that allows efficient write of log records by a partitioned index. However, probes on log require inspecting multiple partitions to restore a page. This approach can delay recovery. The number of partitions can be reduced by intermediate merges. However, this process can interfere with the transaction processing performance. Our log organization is simpler and writes/reads records efficiently. Transactions must wait only for writes on the sequential log to commit. The log indexing does not interfere with transaction processing since records are indexed asynchronously to transaction commit. During recovery, only one fetch on B-tree can restore a tuple individually.

As it was exemplified in Figure 3, transactions Tx1, Tx2, and Tx3 generated log records for updates performed in tuples Tp1, Tp2, and Tp3. Sequential log (Figure 3 (a)) stores the records flushed by the three transactions. The log records with LSN 11, 13, 16, and 19 represent the last update performed in tuple Tp1, for example. A fetch on the indexed log can retrieve the records of LSN 11, 13, 16, and 19 to redo the tuple Tp1. The absence of an index implies the necessity of a full scan on the sequential log to restore Tp1.

4.3 The Recovery Algorithm

After a system failure, the system should initiate the database recovery by restoring tuples through the indexed log. However, the record indexing process is asynchronous to the transaction commit. As a result, some records on the sequential log may not have been indexed before a failure. Therefore, immediately before starting recovery, the system must verify if any records have not yet been indexed. Indexer component must index those records to ensure the recovery consistency. When this process ends, recovery can begin and new transactions can be performed. Thus, the Restorer component begins redoing tuples by traversing the indexed log B-Tree. Each visit to a B-Tree node can retrieve the update records to redo a tuple. After visiting all B-Tree nodes, all database tuples are restored, and the recovery process is completed.

The indexed log recovery scheme can naturally support availability since new transactions can be executed immediately after restoring their required tuples. Furthermore, this recovery scheme can service new transactions whose necessary tuples have not yet been loaded into memory during recovery. When a transaction requires tuples, the system checks if the tuples are stored in memory. If they are not in memory, the Scheduler component must request the Restorer for these tuples on-demand. Then, the recovery manager should pause the incremental recovery (the traversing in B-tree) and begin fetching the necessary tuples for the transaction from the indexed log. After the transaction's tuples are restored, they are marked as restored, the transaction can run, and the system can continue the incremental recovery.

4.4 The Evaluation Prototype

The instant recovery approach proposed in this paper was implemented in Redis 5.0.7 [16] to evaluate the feasibility of indexing for log replay. The evaluation prototype can be downloaded[1]. Redis is an open-source in-memory data structure store used as an in-memory key-value database. Redis is written in ANSI C.

Persistence in Redis can be achieved through snapshotting and logging. In snapshotting, the database is asynchronously transferred from memory to secondary storage at regular intervals as a binary dump using the Redis RDB Dump File Format.

[1]https://drive.google.com/drive/folders/1LTbtY36O0kWIpxZBM-hc1BPvIjICuy2F

In logging, a record of each operation that modifies the database is added to an append-only file (AOF). Redis can automatically rewrite the AOF in the background when it gets too big [16]. Our prototype uses only the AOF, i.e., the RDB was disabled. Moreover, the system does not rewrite the log.

During transaction performing, each update operation generates a log record that contains basically its command, key, and value. For example, the operation *SET(K1, V1)* stores the value *V1* with the key *K1* and generates the log record fields *SET*, *K1*, and *V1*. Each record is written in the AOF atomically only at a committed time and at the same order each in which the command was performed. Our prototype uses the AOF from Redis, i.e., we did not need to implement a sequential log.

The records must be copied from the sequential log to the indexed log periodically. The indexed log is a B-tree implemented in Berkeley DB 4.8 [14]. Berkeley Database (Berkeley DB or BDB) is a software library intended to provide a high-performance embedded database for key/value data.

5 EVALUATION

We have empirically evaluated the instant recovery approach proposed in this research. We used the Memtier Benchmark to perform the tests in Redis. All experiments shown in this paper were executed with 4 worker threads on Intel Core i7-9700k CPU 3.60GHz x 8. The system has 64GB of RAM and 400GB of SSD Kingston SA400S37 as a persistent storage device. The operating system was Ubuntu Linux 18.04.2 LTS.

5.1 Mentier Benchmark

Memtier is a high-throughput benchmarking tool for Redis developed by Redis Labs. This tool has a command-line interface that provides a set of customization and reporting features to generate various workload patterns. It can launch multiple worker threads, with each thread driving a configurable number of clients. The tool can control the ratio between read and write operations. Moreover, it offers control over the pattern of keys used by the operations (e.g., random and sequential patterns). Memtier provides options to set the number of total requests per client or the number of seconds to run a test. The tool offers other options for configuring custom workloads [1, 11]. Memtier has already been used in several scientific works, such as in [25] and [3].

5.2 Recovery Experiments

The first group of experiments was focused on measuring the time to fully recover a database, availability to process transactions after a system failure, time to run a workload entirely, and logging overhead. These experiments were performed on a database containing 99, 507 kyes that generated an 11.8GB sequential log file containing 160 million records. Additionally, an indexed log was generated along with this sequential log using the recovery technique proposed in this work. For each experiment, the system was shut down to simulate a failure. At the database restart, as soon as the recovery process was been triggered, a workload would be submitted. Thus, one could measure transaction throughput and recovery time from system restart.

The key goal was to compare the proposed instant recovery approach to the traditional main memory database recovery. However, we also tested our instant recovery scheme in different scenarios to confirm the following expectations about our technique: (1) an indexed log must be employed to incrementally and on-demand recover the database, and (2) asynchronous indexing of log records must be used to avoid transaction processing overhead. Thus, the experiments have been conducted in the three following scenarios: *(i)* Sequential Log Recovery - SLR; *(ii)* Asynchronous Indexed Log Instant Recovery - AILIR; *(iii)* Synchronous Indexed Log Instant Recovery - SILIR.

The SLR scenario (traditional recovery) uses only a sequential log. In this scenario, transaction update records are written to a sequential log file during transaction processing. The recovery process recovers the database by scanning the entire log file. Transactions can be performed only after the recovery is complete. The AILIR scenario (our approach) uses a sequential log + indexed log. In AILIR, transaction update records are written in a sequential log during transaction processing and stored asynchronously to transaction commit in an indexed log. The SILIR scenario (scenario derived from AILIR) uses only an indexed log. In SILIR, transaction update records are written directly to an indexed log synchronously to the transaction commit. After a failure, for both scenarios ii (AILIR) and iii (SILIR), the recovery manager must traverse the B-tree to recover the database, and transactions can perform during recovery. The SILIR scenario was created to measure the log indexing overhead during transaction processing and instant recovery processing.

For each scenario mentioned above, three experiments were performed by different types of workload: *(i)* read-only workload that contains only read operations, *(ii)* read-write workload that has read and write operations in the 5:5 ratio, and *(iii)* write-only workload that has only write operations. These three workloads were simulated using Memtier benchmark that used 4 worker threads, with each thread driving 50 clients. Each client made 170,000 requests in a random pattern.

Figure 4 shows the results of recovery experiments for the three scenarios (SLR, AILIR, and SILIR) performing the read-only workload, denoted Scenarios Read-Only. The vertical dashed lines in the figure indicate the final recovery time of the respective color approach. AILIR and SILIR recovered the database at the same time interval (85 seconds) since both techniques use the same algorithm to recover the database. They recovered before SLR which took 91 seconds to recover. In addition, they were available for new transactions since the database restart and before SLR which can execute transactions only after full recovery. Those two approaches had a quite similar throughput during the workload performing. Moreover, after the recovery, the three scenarios had a similar throughput. This was because there were no records to flush to the log file. AILIR and SILIR had a slightly lower throughput during recovery due to access to the indexed log. AILIR and SILIR performed the entire workload before SLR, as they can process new transactions during recovery.

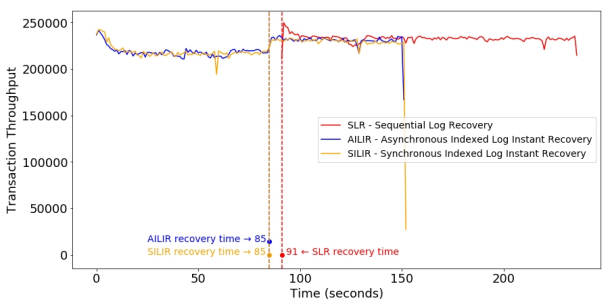


Figure 4: Recovery experiments - Scenarios Read-Only.

The results of recovery experiments for the three scenarios mentioned in this paper performing the read-write workload (denoted Scenarios Read-Write) are in Figure 5. The vertical dashed lines in the figure indicate the final recovery time of the respective color approach. The AILIR approach did not overload the throughput of transactions since its throughput was similar to that of the default approach (SLR). This result was already expected because AILIR and SLR flush log records to secondary memory in a similar manner, except that AILIR additionally indexes the log records. However, the indexing did not interfere with the transaction throughput because it is performed asynchronously to transaction commit. SILIR had the worst performance due to its synchronous log indexing, i.e., a transaction must wait for indexing to confirm its writes. Although SLR recovered the database before AILIR, AILIR was the fastest approach to finish the workload execution. This result was achieved because AILIR has asynchronous indexing and can process transactions while the system is recovering. In addition, the client application did not notice the AILIR recovery, giving the impression that the recovery was instantaneous.

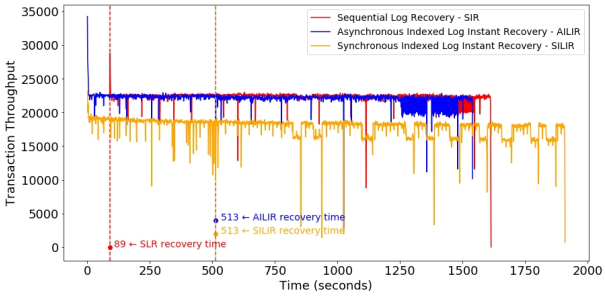


Figure 5: Recovery experiments - Scenarios Read-Write.

Figure 6 presents the results of recovery experiments for the three scenarios performing the write-only workload (denoted Scenarios Write-Only). These results are similar to those in Figure 5. Except for the fact that SILIR recovered the database faster than AILIR. However, this fact did not influence AILIR's performance. The AILIR approach had better performance since it was the fastest approach to finish the workload execution without overloading the transaction throughput. It had a very similar throughput to default recovery.

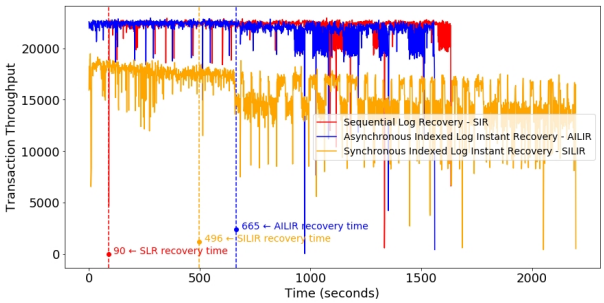


Figure 6: Recovery experiments - Scenarios Write-Only.

5.3 Scalability Experiments

We ran further experiments in which the proposed recovery strategy deals with different log file sizes. The goal is to observe the behavior and performance of the recovery strategy when the log file increases in size. These experiments were performed on the following four databases:

(1) DB1: containing 49, 866 kyes that generated a 5.9GB sequential log file containing 80 million records.
(2) DB2: containing 99, 507 kyes that generated an 11.8GB sequential log file containing 160 million records.
(3) DB3: containing 198, 067 kyes that generated a 23.6GB sequential log file containing 320 million records.
(4) DB4: containing 392, 041 kyes that generated a 47.3GB sequential log file containing 640 million records.

These four databases mentioned above have a ratio of approximately 1:1600 between keys and log records. Each sequential log was generated along with an indexed log using the recovery technique proposed in this work. The experiments used the same scenarios handled in the previous section (Section 5.2): *(i)* Sequential Log Recovery - SLR; *(ii)* Asynchronous Indexed Log Instant Recovery - AILIR; *(iii)* Synchronous Indexed Log Instant Recovery - SILIR. An experiment was performed for each of the four databases (DB1, DB2, DB3, and DB4) in each scenario (SLR, AILIR, and SILIR). For each experiment, the system was shut down to simulate a failure. At the database restart, as soon as the recovery process was been triggered, a workload would be submitted. The workload was simulated using Memtier benchmark that used 4 worker threads, with each thread driving 50 clients. Each client made 300,000 requests. The workload had 5:5 read and write operations in a random pattern. From the system restart, we measured the average transaction throughput during the execution of the workload, the recovery time, and the total workload execution time.

Figure 7 presents the recovery time obtained in each test. As expected, these results show that the number of database keys and log records can delay the recovery process in all approaches, as the recovery time increases with the size of the database. The instant recovery approaches (AILIR and SILIR) had a similar recovery time because they use the same recovery technique. However, AILIR was slightly faster than SILIR in all tests. This is because the weight of synchronous indexing by SILIR influences the overall performance of the system. SLR recovered the database faster than AILIR and SILIR.

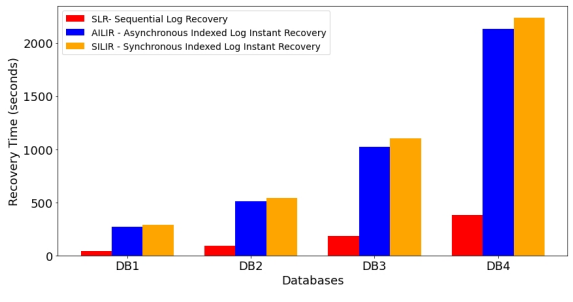


Figure 7: Scalability experiments - Recovery time.

Figure 8 shows that AILIR interferes very little in the throughput of transactions. This is evidenced by the fact that the AILIR average throughput, for the execution of a given workload, remained similar to that of the standard approach (SLR) in all experiments. On the other hand, SILIR's average transaction throughput was much lower than that of AILIR, proving that asynchronous indexing is essential for a better performance of

the instant recovery technique. The throughput difference between AILIR and SLR may occur because the Indexer component blocks the sequential log to read it. In the meantime, transactions must wait for the lock to be released before writing records to the log. Nevertheless, the approach proposed in this paper has the advantage of high availability. It can perform new transactions since the system restart, while the standard approach must wait for full database recovery to perform new transactions.

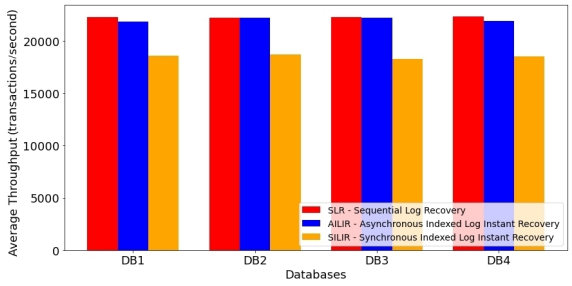


Figure 8: Scalability experiments - Average throughput.

The results of the experiments in Figure 9 show that the execution time of AILIR's workload is slightly longer than that of SLR. Although the experiments in Figures 4, 5, and 6 have shown that AILIR performed the workload faster than SLR, the experiments in Figure 9 show that this behavior changes with higher workloads. The workload of the experiments in Figure 9 (300,000 operations) is 1.7x greater than the workload of the experiments in Figures 4, 5, and 6 (170,000 operations). This is because SLR's transaction throughput is slightly higher than that of AILIR, as shown in Figure 8.

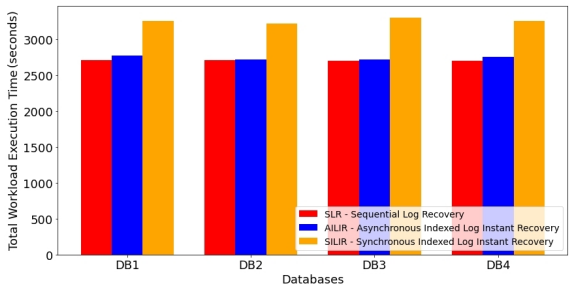


Figure 9: Scalability experiments - Total workload execution time.

6 CONCLUSION

This paper proposed an instant recovery approach for main memory database systems. The proposed approach allows new transactions to run concurrently to the recovery process. Our approach implements an indexed log to fetch tuples directly on the log to restore data incrementally. Consequently, new transactions are scheduled as soon as required tuples are restored into the main memory database. Furthermore, the proposed recovery mechanism restores data on-demand since it restores tuples for new transactions whose data has not yet been restored.

The results show that instant recovery reduces the perceived time to repair the database since transactions can be performed since the system is restarted. In other words, it can effectively deliver tuples that new transactions need during the recovery process. The experiments also analyzed the impact of using a log indexed structure on transaction throughput rates in an OLTP workload benchmark.

REFERENCES

[1] Memtier Benchmark. 2020. GitHub - RedisLabs/memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool. Retrieved August 26, 2020 from https://github.com/RedisLabs/memtier_benchmark

[2] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. 2011. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. Association for Computing Machinery (ACM), 265–276.

[3] Wenqi Cao, Semih Sahin, Ling Liu, and Xianqiang Bao. 2016. Evaluation and analysis of in-memory key-value systems. In *2016 IEEE International Congress on Big Data (BigData Congress)*. IEEE, 26–33.

[4] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. 1984. *Implementation techniques for main memory database systems*. Vol. 14. Association for Computing Machinery.

[5] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1243–1254.

[6] Margaret H Eich. 1986. Main memory database recovery. In *Proceedings of 1986 ACM Fall joint computer conference*. IEEE Computer Society Press, 1226–1232.

[7] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin Levandoski, Thomas Neumann, Andrew Pavlo, et al. 2017. Main Memory Database Systems. *Foundations and Trends® in Databases* 8, 1-2 (2017), 1–130.

[8] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. *ACM Sigmod Record* 40, 4 (2012), 45–51.

[9] Florian Funke, Alfons Kemper, Tobias Mühlbauer, Thomas Neumann, and Viktor Leis. 2014. HyPer Beyond Software: Exploiting Modern Hardware for Main-Memory Database Systems. *Datenbank-Spektrum* 14, 3 (2014), 173–181.

[10] Le Gruenwald, Jing Huang, Margaret H Dunham, Jun-Lin Lin, and Ashley Chaffin Peltier. 1996. Recovery in main memory databases. (1996).

[11] Redis Labs. 2020. Redis Labs | The Best Redis Experience. Retrieved October 06, 2020 from https://redislabs.com

[12] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory oltp recovery. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE, 604–615.

[13] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[14] Michael A Olson, Keith Bostic, and Margo I Seltzer. 1999. Berkeley DB.. In *USENIX Annual Technical Conference, FREENIX Track*. 183–191.

[15] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[16] Redis. 2020. Redis. Retrieved August 26, 2020 from https://redis.io

[17] Caetano Sauer. 2017. *Modern Techniques for Transaction-oriented Database Recovery*. Ph.D. Dissertation. University of Kaiserslautern, Kaiserslautern, Germany.

[18] Caetano Sauer, Goetz Graefe, and Theo Härder. 2017. Instant restore after a media failure. In *Advances in Databases and Information Systems*. Springer, 311–325.

[19] Caetano Sauer, Goetz Graefe, and Theo Härder. 2018. FineLine: log-structured transactional storage and recovery. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2249–2262.

[20] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[21] Kian-Lee Tan, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. 2015. In-memory databases: Challenges and opportunities from software and hardware perspectives. *ACM SIGMOD Record* 44, 2 (2015), 35–40.

[22] Yingjun Wu, Wentian Guo, Chee-Yong Chan, and Kian-Lee Tan. 2017. Fast Failure Recovery for Main-Memory DBMSs on Multicores. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 267–281.

[23] Chang Yao, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, and Sai Wu. 2016. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1119–1134.

[24] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948.

[25] Yiying Zhang and Steven Swanson. 2015. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–10.

[26] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*, Vol. 14. 465–477.

Short Paper

HorsePower: Accelerating Database Queries for Advanced Data Analytics

Hanfeng Chen, Joseph V. D'silva, Laurie Hendren, Bettina Kemme
McGill University
hanfeng.chen@mail.mcgill.ca,joseph.dsilva@cs.mcgill.ca,hendren@cs.mcgill.ca,kemme@cs.mcgill.ca

ABSTRACT

The rising popularity of data science has resulted in a challenging interplay between traditional declarative queries and numerical computations on the data. In this paper, we present and evaluate the advanced analytical system HorsePower that is able to combine and optimize both programming styles in a holistic manner. It can execute traditional SQL-based database queries, programs written in the statistical language MATLAB, as well as a mix of both by supporting user-defined functions within database queries. HorsePower exploits HorseIR, an array-based intermediate representation (IR), to which source programs are translated, allowing to combine query optimization and compiler optimization techniques at an intermediate level of abstraction.

1 INTRODUCTION

Complex data analytics has become the cornerstone of our data-driven society. Although the amount of data stored in traditional relational database systems (DBS) has been growing rapidly, the by far most common current approach is to take the data first out of the DBS and load it into stand-alone analytical tools, which are based on languages such as Python, or the statistical languages MATLAB [1], and R [3]. However, as the size of the data increases, the expensive data movement between DBS and analytics tools can become a severe bottleneck.

Integrating analytical capabilities into the DBS avoids such expensive data exchange. A common approach is to use user-defined functions (UDFs) that are embedded in SQL queries [13]. For example, MonetDB supports UDFs written in Python, that are executed by a Python language interpreter that is embedded inside the DBS engine.

While no data transfer is needed with this approach, there are still two separate execution environments, one being the SQL execution engine, the other the programming language execution environment. This can lead to costly data format conversion. Furthermore, the SQL and the UDF components of the query are each individually optimized by their respective execution environments, without the consideration of any holistic optimization across the entire task.

To address these issues, we propose `HorsePower`, an advanced analytical SQL system, which provides a holistic solution to integrate UDFs in SQL queries. The system is based on HorseIR [5], an array-based intermediate representation (IR) language which was developed to explore the usage of compiler optimizations for query execution. Chen et al. [5] translated the execution plans of standard SQL queries into HorseIR and compiled the generated HorseIR code using various compiler optimization strategies developed for array-based languages. Using arrays to represent database columns, HorseIR follows conceptually the data model of column-based DBS, which has been proven to be effective for data analytics tasks.

HorsePower extends the idea to a full-fledged execution environment for data analytics. Additionally to supporting plain SQL queries, HorsePower also supports functions written in MATLAB, a popular high-level array language widely used in the field of statistics and engineering. HorsePower can take stand-alone functions written in MATLAB and translate them to HorseIR, of have these functions be embedded in SQL queries and then translate all into a single HorseIR program, before optimizing and compiling the code in a holistic manner.

As such HorsePower avoids the overhead of inter-system data movements as it has a single execution environment, and eliminates the barriers between SQL queries and analytical functions allowing optimizations across both the declarative and functional parts of the query.

The contributions of this paper are thus as follows:

- We present HorsePower, an advanced analytical system, that extends the approach proposed in [5] to not only offer a compiler-based execution environment for SQL queries, but also for programs written in the array-based language MATLAB and for SQL queries with embedded UDFs.
- HorsePower uses a holistic approach of exploiting array-based compiler optimization techniques for both SQL and MATLAB taking advantage of the conceptual similarities of columns and arrays.
- The performance of HorsePower is shown through an extensive set of experiments on programs written in MATLAB, and SQL queries with embedded UDFs.

2 BACKGROUND

2.1 HorseIR: an Array-based IR for SQL

Recent years have seen the development of modern query compilers that translate an SQL query into an intermediate representation (IR) before target code is generated from the IR, making it possible to leverage any existing code optimizations available within the IR platform.

In this context, HorseIR [5] was developed as a high-level IR specifically for database applications [7]. Being an array-based IR, it is relatively straightforward to generate basic HorseIR code following the execution plans developed by column-based DBS, as the operators executing on entire columns can be translated to functions executing on vectors in HorseIR. In fact, Chen et al. [5] took the execution plans generated by the column-based database system HyPer [11], that incorporate a wide range of traditional DBS optimizations, as the input for generating HorseIR programs.

In this regard, HorseIR provides a rich set of array-based built-in functions to which one can map the standard database operations. Moreover, the HorseIR compiler provides vital optimizations over these array-based operations. For example, *loop fusion* merges multiple loops into one loop, allowing for an intuitive merge of chained operations and

```
SELECT SUM(l_price * l_discount) AS RevenueChange
FROM lineitem WHERE l_discount >= 0.05;
```

```
module ExampleQuery{
 def main(): table{
  ...
  // assume t1, t2 are references to l_price/l_discount columns
  t3:bool = @geq(t2, 0.05);
  t4:f64  = @compress(t3, t1);
  t5:f64  = @compress(t3, t2);
  t6:f64  = @mul(t4, t5);
  t7:f64  = @sum(t6);
  ...}}
```

Figure 1: Example query and its HorseIR program

thus, avoiding intermediate results. Thus, optimizations developed for array-based programming languages can be exploited to improve query performance.

Example The top of Figure 1 shows a simplified version of Query 6 of the TPC-H benchmark [16] computing the change in total revenue given prices and discounts from the table `lineitem`. A basic translation into a HorseIR program prior to performing any optimizations is shown at the bottom of Figure 1, outlining only the part of the code that performs the actual relational operators. We assume that arrays `t1` and `t2` represent the price and discount columns. The program computes the WHERE condition (`@geq`), which returns a boolean vector of the same length as `t2` with `true` values in all rows that fulfill the condition. The function `@compress` then extracts from both `t1` and `t2` the rows for which the boolean vector has a `true` value. The output are "compressed" vectors with relevant rows, over which then the aggregation is performed in two steps.

HorseIR Optimizations As can be seen, such an approach can generate a fair amount of intermediate results (arrays `t3` to `t6` in the example). If lines 5 to 9 are translated to lower-level code independently, each of them generates its own `for` loop over the corresponding arrays. However, array-based optimization techniques, including loop fusion, and some pattern-based optimizations developed specifically for the operator sequences found in SQL statements, allow the HorseIR compiler to fuse these loops to just one loop to avoid materializing these intermediate vectors. To do such fusion, HorseIR first builds a data dependence graph across all the statements. Statements which can be fused or follow a pattern, are then identified by a well-defined data flow analysis, and compiled together to efficient C code. For our example, the resulting sequential C code would look similar to

```
...
revenue = 0;
for(i = 0; i < numRows; i++)
{ if(t2[i] >= 0.05) revenue += t1[i] * t2[i]; }
...
```

Although the example C code does not convey it explicitly, behind the scenes, HorseIR uses OpenMP to compile the program into a parallel implementation, as outlined in [5].

2.2 Traditional Database UDFs

A UDF is a high-level language function embedded within an SQL statement, and is used to offload partial computation into are more concise language than SQL, or provide additional functionality. To support UDFs, the database system integrates the language runtime environment into the DBS (such as the Python interpreter in MonetDB [13]). We will focus only on *Scalar UDFs* and *Table UDFs*, as

```
FUNCTION RevChangeSclr(price,discount)
  RETURN price * discount;
END
```

```
SELECT SUM(RevChangeSclr(l_price,l_discount)) AS RevChange
FROM lineitem WHERE l_discount >= 0.05;
```

Figure 2: Rewriting the example query with a scalar UDF

these are the most commonly employed types of UDFs and also the ones supported presently in HorsePower.

A *scalar UDF* returns a single value per row (which could be a vector) and can be therefore essentially used wherever a regular table column is used, such as the `SELECT` or the `WHERE` clause of SQL queries. Figure 2 shows a scalar UDF which performs the multiplication that was originally part of the `SELECT` clause in Figure 1. In a column-based database system, the execution of such a query first evaluates the `WHERE` clause on `l_discount`, returning a boolean vector. Then, the database applies the corresponding boolean selection on columns `l_discount` and `l_extendedprice`, returning compressed vectors containing the rows where the boolean vector was true. These columns are then given to the UDF as arrays, and the UDF performs an element-wise multiplication on them and produces a result array. This is then the input to the `SUM` operator. Thus, the UDF is only called a single time and works on entire arrays.

A *table UDF* returns a table-like data structure, and thus, is typically called within the `FROM` clause of an SQL statement, similar to regular database tables. For an example of a table UDF, we refer to a technical report [6].

Introducing UDFs into queries can bring performance issues. If the data types used by the two execution environments are different, this can introduce a conversion overhead when exchanging data. Further, as UDF languages are typically black-boxes to the database engine, cross optimization attempts are minimal, resulting in sub-optimal execution plans.

3 HORSEPOWER

In this section we present HorsePower, a system designed for the code generation and optimization of HorseIR generated from (1) SQL queries, (2) MATLAB programs, and (3) SQL queries with analytical functions written in MATLAB.

3.1 SQL to HorseIR

While prior work used HyPer's execution plans [11] to translate SQL to HorseIR, HorsePower uses MonetDB's execution plans, as MonetDB supports UDFs and the execution plans contain the relevant UDF information. Our implementation first translates the tree-based plans to JSON objects that are then translated to HorseIR[1].

Furthermore, HorsePower supports a wider range of SQL queries than [5], which did not properly support multi-join queries. This includes all queries of the TPC-H benchmark [16].

3.2 MATLAB to HorseIR

MATLAB is a sophisticated dynamic language which provides numerous flexible language features. In order to transform MATLAB code to HorseIR, as an intermediate step,

[1]HorsePower could generate it own execution plans. However, as the traditional query optimization techniques are not the focus of our research, we preferred to integrate the already optimized execution plans generated by existing DBS.

HorsePower calls upon the McLab framework [2] which translates MATLAB programs to its own internal IR, called TameIR, handling MATLAB's many dynamic features and lack of strict typing. Type and shape information for all variables in the program are automatically derived. Furthermore, class program analysis steps, such as constant propagation, are performed to produce optimized TameIR code [9]. TameIR can represent MATLAB's matrix and high-dimension arrays, and currently supports an essential subset of MATLAB array operations.

HorsePower then translates TameIR code to HorseIR. So far, this translator supports a core subset of MATLAB features and built-in functions. It preserves MATLAB pass-by-value semantics but automatically switches to pass-by-reference when it determines that the input parameters are not modified, avoiding data copies. It supports the common control structures `if-else` and `while` with a restriction on the condition which must be a single boolean element. While explicit loop iteration is not supported, MATLAB's array-based built-in functions (which have implicit loop execution) are translated in a straightforward way as similar functions exist in HorseIR. All types supported by TameIR are also supported by HorseIR, however, due to type rule mismatches, input types for some operators are restricted (e.g. because *integer + double* returns integer in MATLAB, but double in HorseIR). Finally, the translator requires MATLAB arrays to have the data layout of `1-by-N` instead of `N-by-1`, as the former one is more cache-friendly in MATLAB.

3.3 SQL and UDF to HorseIR

HorsePower supports SQL queries with embedded UDFs written in MATLAB. As described in Section 3.1, HorsePower uses execution plans generated by MonetDB, which contain hooks into UDFs with their names, and input and output parameters, but otherwise treat the UDFs as a black-box. HorsePower translates such a plan to HorseIR, where the invocation of the UDF is translated to a method invocation in HorseIR. Next, we generate a separate piece of HorseIR code by translating the UDF written in MATLAB using the MATLAB-to-HorseIR translator introduced in Section 3.2. Finally, the two segments of code for SQL and UDFs are integrated into a single HorseIR program.

HorsePower supports both scalar and table UDFs. In order to make the MATLAB functions conform to the semantic form expected of these types of UDFs, we enforce some restrictions on the MATLAB functions. For instance, we require a function to have one return statement with either a single vector (for scalar UDFs) or a table-like data structure (for table UDFs).

Figure 3 shows the HorseIR program for the example query in Figure 2 with a scalar UDF. The HorseIR code consists of a module with two methods: the SQL component is translated to the main method, and the UDF is translated to the method `RevChangeSclr` which takes two arrays of type float as input and returns the resulting product. This method is called by the main method, which otherwise is the same as we have already seen in Figure 1.

3.4 Holistic HorsePower Optimizations

HorsePower performs compiler-based optimizations when translating a HorseIR program to target C code. We have discussed in Sec. 2.1, how *automatic loop-fusion* and *pattern-based*, as introduced in [7] lead to efficient parallel C code.

```
1  module ExampleQuery{
2    def RevChangeSclr(price:f64, discount:f64): f64{
3       x0:f64 = @mul(price, discount); // S5
4       return x0;
5    }
6    def main(): table{
7       ...
8       // compute revenue change
9       t3:bool= @geq(t2, 0.05:f64);     // S0
10      t4:f64 = @compress(t3, t1);      // S1
11      t5:f64 = @compress(t3, t2);      // S2
12      t6:f64 = @RevChangeSclr(t4,t5); // S3
13      t7:f64 = @sum(t6);               // S4
14      ...
15 }}
```

Figure 3: HorseIR code for the Query in Figure 2

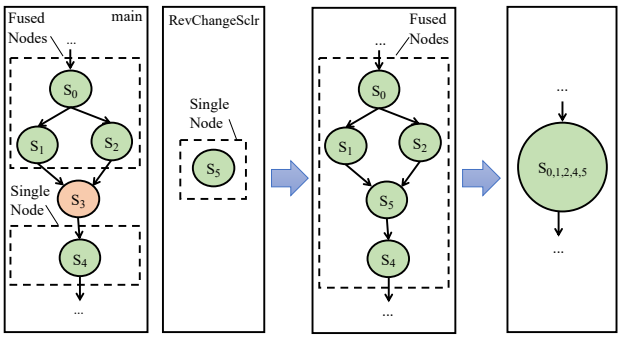


Figure 4: Dependence graphs for the example in Figure 3

However, such optimizations require all statements to be in one method. But when SQL statements have embedded UDFs, the HorseIR code has at least two methods, with a main method calling the method representing the UDF as shown in our example in Figure 3.

If we were to optimize both parts independently using loop fusion and pattern-based fusion, the overall result would be sub-optimal. In fact, if we look at the dependence graph for this program on the left side of Figure 4 (with S_0 to S_4 depicting the statements in the code), we can see that the optimization opportunities are now separated into three snippets: before, after, and in the method being called in the statement S_3. The snippets have to be optimized individually because the content of the statement S_3 is invisible to the rest of the code. Thus, statements S_1 and S_2 of the main method need to be evaluated and intermediate results `t4` and `t5` cannot be eliminated as the method `RevChangeSclr` requires their actual values to be passed as parameters. Furthermore, the return value of the method needs to be materialized to be assigned to `t6` which is then the input of the statement S_4. This means the potential scope for fusion is significantly reduced leading to more intermediate results.

In order to enable a more holistic cross-optimization, we use the concept of *inlining*. This involves replacing the method calls within the main method with the corresponding code segments that constitute the method that is being called. For our example program in Figure 3 this means the code of `RevChangeSclr` can be inlined into the main method with the generated HorseIR being almost the same as the one in Figure 1 except for possibly different variable names. As a result, a dependence graph can be built across the main method, as illustrated on the right side of Figure 4, allowing for loop fusion across all statements and generating a single loop of all tasks as outlined in Section 2.1, and avoiding the

materialization of any intermediate results introduced by UDF invocations.

In some scenarios method inlining offers additional optimization opportunities, such as the elimination of unused computations. For example, consider a scenario where a table UDF computes and returns two columns as part of its invocation, but the enclosing SQL query itself uses only one of those two columns. HorsePower will employ the *backward slicing* technique [15] to avoid the computation of the unused column in the table UDF.

While performing inlining, to respect the pass-by-value convention for parameter passing, a copy of the object used as the parameter will be generated if the parameter is found to be modified inside the original callee method. This ensures that inlining does not result in any unintended data modifications to the objects inside the method that was making the call. Further, if inlining results in any variable name conflicts, they are resolved by assigning new but unique variable names. Finally, an inlined method is removed if it can be inlined in all the code locations where it is called.

4 EVALUATION

In this section we present the evaluation result of our framework for pure MATLAB programs, and for SQL queries with analytical UDFs written in MATLAB. For the latter, we compare it with MonetDB.

The experiments are conducted on a server equipped with 4 Intel Xeon E7-4850 2.00GHz (total 40 cores with 80 threads, and 24 MB of shared L3 CPU cache) and 128 GB RAM running Ubuntu 18.04.4 LTS. We use GCC v8.1.0 to compile HorseIR source code with optimization options `-O3` and `-march=native`; MonetDB version v11.35.9 (Nov2019-SP1) and NumPy v1.13.3 along with Python v2.7.17 interpreter for embedded Python support in MonetDB; and MATLAB version R2019a.

The response time is measured only for the core computation, and excludes the overhead for parsing SQL, plan generation, compilation, and serialization for sending the results to the client. We only consider execution time once data resides in the main memory. We run each test 15 times but only measure the average execution time over the last 10 times. Scripts and data used in our experiments can be found in our GitHub repository[2].

4.1 MATLAB Benchmarks

We first evaluate MATLAB programs in order to understand the performance of using HorsePower for executing non-SQL based data analytics, and use the following benchmarks: the `Black-Scholes` algorithm from the PARSEC benchmark suite v3.0 [4] having two UDFs *BlackScholes* and *CNDF*, and the `Morgan` algorithm [8] from a finance application having a main function *morgan* and another function *msum*. Both contain several element-wise functions and are fully vectorizable.

In our experiments, we compare the following:

- We execute the original MATLAB program using the MATLAB interpreter with default settings.
- We compile the HorseIR program generated from the MATLAB code into C code without any of the optimizations that we mentioned in Section 3.4. We refer to this version as HorsePower-Naive. As such, it is likely to produce a similar amount of intermediate results as the MATLAB interpreter.

[2] https://github.com/Sable/edbt21-analysis

Table 1: Speedup of HorsePower over MATLAB in execution time using Black-Scholes (in milliseconds)

Size	MATLAB	HorsePower			
		Naive	Speedup	Opt.	Speedup
1M	61	66	0.92x	7	9.34x
2M	145	137	1.06x	14	10.17x
4M	491	463	1.06x	49	10.12x
8M	1009	1384	0.73x	117	8.60x

- We compile the HorseIR code into C code with all optimizations enabled, referred to as HorsePower-Opt.

Table 1 shows the execution times for MATLAB and for the two HorsePower versions with different sizes of the Black-Scholes tables. We also indicate the speedup of HorsePower over MATLAB in execution time. Note that the MATLAB interpreter uses all physical threads. For HorsePower, we used 40 threads.

The execution times for MATLAB and HorsePower-Naive are similar, with slightly better performance for MATLAB, probably due to MATLAB having more efficient library functions. When comparing with HorsePower-Opt, MATLAB is significantly slower. The reason is that HorsePower-Opt optimizations, in particular loop fusion, are able to avoid many intermediate results. We also observe that the size of the data set plays a minor role.

For Morgan (no table shown due to space limitations) we run experiments up to 8 million rows as well. HorsePower-Naive also provides similar performance to MATLAB with smaller data sizes, but already has a speedup of 2 with 8 million rows. We believe the reason is our efficient parallel implementation of built-in functions, such as the cumulative sum. Again, the optimized version is significantly faster, with a speedup of 7 with 8 million rows.

In summary, HorsePower can execute data analytics tasks in an efficient manner due to its data-centric IR and compiler optimization techniques.

4.2 SQL and UDF Benchmarks: TPC-H

This is the first of two sections to evaluate the performance of HorsePower in executing SQL statements with embedded UDFs, and comparing it with MonetDB.

Froid [14] proposed a whole range of queries derived from the TPC-H benchmark in which part of the SELECT or WHERE clauses, e.g., to check certain conditions, are outsourced into a UDF. In all cases, these are scalar UDFs. For instance, they propose a variation of the q6 of the TPC-H benchmark, which is very similar to our example query of Figure 1, simply containing more conditions.

For MonetDB, we rewrote the queries to use Python-based UDFs, for HorsePower, the UDFs are written in MATLAB. The structure of the programs is very similar for both languages. Some of the proposed UDFs have embedded SQL statements which are currently not supported by the McLab framework that we use. Thus, we excluded those unsupported queries and present results only for queries q1, q6, q12, q14, and q19.

Table 2 shows the execution times of these queries with a different number of threads using HorsePower and MonetDB. When first looking only at MonetDB we can see that execution times are relatively low for some queries and improve with an increasing number of threads considerably (q1 and q14), but are high for others with little benefit of parallelization (q6, q12, q19). The reason is that in these queries, the UDF is in the WHERE clause and MonetDB

Table 2: Speedup (SP) of HorsePower over MonetDB in execution time using the modified TPC-H benchmarks with UDFs

Thread	MonetDB (ms)					HorsePower (ms)									
	q1	q6	q12	q14	q19	q1	SP	q6	SP	q12	SP	q14	SP	q19	SP
T1	16853	48832	137195	1040	69045	3799	4.44x	392	125x	900	152x	904	1.15x	858	80.5x
T8	5724	47775	143714	773	72124	3316	1.73x	56	853x	300	479x	396	1.95x	364	198x
T32	2502	44636	140438	750	64267	1883	1.33x	45	1000x	170	826x	216	3.48x	209	307x

has to perform costly data conversion when sending the entire database columns as arrays to the Python interpreter in order to execute the UDF. MonetDB is able to use zero-copy transfer for data types where the database system uses the same main-memory representation as Python. But for strings, it needs to convert the data to a different format as the database internal and the Python formats are incompatible. This data conversion seems to not be parallelized to multiple threads, making it the predominant factor of the execution. In q1 and q14, the UDFs are in the SELECT clause (where data sizes are smaller as they got reduced due to the selection that was already executed), and do not require any string conversions.

HorsePower has overall much better performance for all queries, being under 1 second for all queries except q1, and can always improve execution times by increasing the number of threads. As no data conversion is necessary it is orders of magnitude faster than MonetDB for queries q6, q12, and q19. We observe the advantage of having a unified execution environment that has translated both the UDF part and the SQL part to a single HorseIR program with its own data structures. But we also observe significant improvements for q1 and q14. These are due to the unified optimization across the HorseIR code generated from SQL and UDF.

4.3 SQL and UDF Benchmarks: MATLAB

In this second experiment, we embed the Black-Scholes algorithm in form of UDFs into SQL queries.

We again have a HorsePower version, with the Black-Scholes UDF implemented in MATLAB, and a MonetDB version, with the UDF implemented in Python UDF using the NumPy library and the same array programming style as the MATLAB UDF.

In order to understand the implication of having the UDFs written in different programming languages, we first compared the execution time of Black-Scholes written in Python and using HorseIR (both naive and optimized). Execution is in one thread because NumPy does not support multi-threading. Similar to what we have seen with our analysis with MATLAB, a naive usage of HorseIR provides similar execution time as Python (around 500 ms); performing optimizations achieves a speedup of 2.

In order to look at the impact of embedding this UDF into SQL statements, we created both scalar and table UDF variations as well as designed several enclosing SQL statements that offer different potential for optimizations. In particular, we created a *scalar UDF* that returns just the computed `optionPrice` to the calling SQL.

```
CREATE SCALAR UDF bScholesUDF(spotPrice, ..., optionType)
{
  import blackScholesAlgorithm as bsa
  return bsa.calcOptionPrice(spotPrice, ..., optionType)
};
```

Furthermore, we implemented the solution as a *Table UDF*, which returns in table form the computed `optionPrice` along with the associated `spotPrice` and `optionType` which are columns from the original input table.

In order to have a broad set of tests and comparisons, we first integrated these two UDF versions into a straightforward base query. From there we created three significant variations of this base query that had different columns in the `SELECT` and `WHERE` clauses. Furthermore, the selectivity of WHERE clause can be high (returning few records) or low (having many qualifying records).

Table 3 shows the result of all the variations for MonetDB and HorsePower for 1 thread (T1) and 64 threads (T64).

Base query. The base query `bs0_base` selects all the data from the database table and passes it to the UDF and returns all the data produced by the UDF.

```
-- Base query, bs0_base, Scalar UDF
SELECT spotPrice, optionType,
  bScholesUDF(spotPrice,...,optionType) AS optionPrice
FROM blackScholesData;

-- Base query, bs0_base, Table UDF
SELECT spotPrice, optionType, optionPrice
FROM bScholesTblUDF ((SELECT * FROM blackScholesData));
```

We first observe that for MonetDB multi-threading has little impact on its performance while HorsePower benefits a lot. As Python is not multi-threaded, the Black-Scholes UDF in MonetDB runs always in a single thread even if 64 threads are enabled, while HorsePower creates optimized parallel also for the Black-Scholes part. But HorsePower is already significantly better with a single thread. We then find that HorsePower has even significant benefits with a single thread. In fact, HorsePower's execution time for the entire query is nearly the same as executing the Black-Scholes algorithm alone, while MonetDB takes nearly double the time (> 900 ms) to execute the entire query than the time used by the Python interpreter to execute Black-Scholes (around 500 ms). The reason for this performance penalty in MonetDB must be the communication between its SQL engine and the Python UDF interpreter.

Variation 1. The first variation `bs1_*` applies a predicate condition on `spotPrice`, a column which is actually part of the input database table. The objective of this test case is to analyze if the systems can intelligently avoid performing the UDF computation on records that will not be in the result set. As can be seen, for one thread, HorsePower's speedup over MonetDB is at least 3.5x for both scalar and table UDFs, and for 64 threads at least 50x. MonetDB follows the traditional database optimization technique of applying high selectivity operations first, discarding the records that do not qualify before processing the UDFs. As HorsePower relies on MonetDB for database execution plans, it is similarly impacted by the plans generated by MonetDB for table UDF based queries. This results in HorsePower's own table UDF based queries costing more than its scalar versions. However, unlike MonetDB, HorsePower benefits from being able to avoid data copies and conversions as well as from generating parallelized code for UDFs, thus expanding this performance gap when the number of threads increases.

Variation 2. In the next variation, `bs2_*`, the SQL does not include the computed column `optionPrice` in the final

Table 3: Performance comparison between HorsePower (HP) and MonetDB (MDB) for variations in Black-Scholes.

UDF	Selectivity	Table UDF (ms)						Scalar UDF (ms)					
		T1			T64			T1			T64		
		MDB	HP	Speedup	MDB	HP	Speedup	MDB	HP	Speedup	MDB	HP	Speedup
bs0_base	100.0%	927.5	249.8	3.71x	774.0	7.09	109x	670.0	249.5	2.69x	696.5	7.06	98.6x
bs1_high	0.2%	926.4	256.2	3.62x	818.0	7.62	107x	6.10	0.32	19.1x	6.55	0.13	50.4x
bs1_low	99.8%	929.7	266.4	3.49x	832.9	14.6	57.0x	725.4	169.6	4.28x	645.4	4.90	132x
bs2_high	0.2%	895.6	4.67	192x	791.5	0.70	1131x	4.29	4.59	0.93x	3.52	0.63	5.59x
bs2_low	99.8%	916.4	11.0	83.7x	820.4	6.64	124x	15.9	10.95	1.45x	5.11	5.95	0.86x
bs3_high	10.0%	911.8	259.0	3.52x	824.4	10.1	81.6x	673.8	179.3	3.76x	623.2	7.69	81.0x
bs3_low	90.0%	879.1	262.5	3.35x	793.6	13.7	57.8x	685.4	182.6	3.75x	641.7	12.8	50.1x

result. A smart system should be able to analyze the semantics of the request and avoid processing the UDF both together. MonetDB is able to do the optimization when the SQL query is using the scalar UDF, avoiding the computation of the `optionPrice` column that is not included in the final result. Similarly, HorsePower, being an integrated system, can avoid the computation of `optionPrice` by using a backward slice. As both avoid executing the UDF, HorsePower has only moderate speedup over MonetDB due to other optimizations. However, with a table UDF, MonetDB is unable to avoid this computation as there is no way for it to pass this optimization information to the UDF interpreter. On the other hand, HorsePower uses method inlining and backward slicing to remove this computation, offering a huge advantage.

Variation 3. The last variation, `bs3_*` applies a predicate condition on `optionPrice`. As this is a column computed by the UDFs, both the systems have to process the UDFs across all input records before discarding records that do not qualify, providing limited opportunities for optimization. As can be seen, HorsePower has speedups of around 3.5x for both scalar and table UDFs with one thread and between around 50x and 80x for 64 threads. HorsePower has better performance than MonetDB simply because HorsePower can avoid the data movement between the UDF. With more threads, HorsePower's speedup is even better as the data movement in MonetDB is not parallelized and takes most of the time in the whole execution pipeline.

In summary, HorsePower avoids the problems of a black-box integration of programming language execution environments as used in current DBS. As such, it avoids expensive data conversions, can optimize in a holistic manner and provides full support for parallelization, leading to significant speedups.

5 RELATED WORK AND CONCLUSIONS

Intermediate representations and compiler techniques have been applied by others to improve the performance of database queries. However, there is little research in these systems extending to support UDFs within the database queries.

Froid [14] shows a holistic optimization solution by transforming simple UDF to relational code. Thus, the existing query optimizer can be utilized for the optimizations of the execution plan. However, this approach is limited as not all UDFs are translatable to a relational operator.

Weld [12] presents its IR (WeldIR) to support the code generation from various source languages. WeldIR is able to handle database queries and call UDFs written in C code. However, in contrast to HorsePower that automatically optimizes across different source languages, such capabilities have not been implemented by Weld.

Lara [10] is a domain-specific language tailored for relational algebra and UDFs. Its code is first compiled to an IR which is able to inspect UDFs by collecting necessary information from UDFs. Thus, Lara can optimize such transparent UDFs together with its IR code. This is different from our HorsePower which compiles database queries and UDFs to its common IR with holistic optimizations enabled.

In conclusion, `HorsePower` differs from previous work in that it is a compiler-based approach exploiting array-based optimizations to support database queries, MATLAB programs and database queries with analytical UDFs in a holistic framework. Given the very promising evaluation results, future work will integrate different programming languages, and enhance our relational operators.

REFERENCES

[1] MATLAB. https://www.mathworks.com. (????). [Last accessed in Feb. 2021].

[2] McLab: A Framework for Dynamic Scientific Languages. http://www.sable.mcgill.ca/mclab/. (????). [Last accessed in Feb. 2021].

[3] R Project. https://www.r-project.org. (????). [Last accessed in Feb. 2021].

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of Intl. Conference on Parallel Architectures and Compilation Techniques*. 72–81.

[5] Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme, and Laurie Hendren. 2018. HorseIR: Bringing Array Programming Languages Together with Database Query Processing. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, (DLS'18)*. 37–49.

[6] Hanfeng Chen, Joseph Vinish D'silva, Laurie Hendren, and Bettina Kemme. 2021. *Accelerating Database Queries for Advanced Data Analytics: A New Approach*. Technical Report SABLE-TR-2021-1. Sable Group, McGill University. http://www.sable.mcgill.ca/publications/techreports/#report2021-1

[7] Hanfeng Chen, Alexander Krolik, Bettina Kemme, Clark Verbrugge, and Laurie Hendren. 2020. Improving Database Query Performance with Automatic Fusion. In *Proceedings of Intl. Conference on Compiler Construction, (CC'20)*. 63–73.

[8] Wai-Mee Ching and Da Zheng. 2012. Automatic Parallelization of Array-oriented Programs for a Multi-core Machine. *International Journal of Parallel Programming* 40, 5 (2012), 514–531.

[9] Anton Willy Dubrau and Laurie Jane Hendren. 2012. Taming MATLAB. In *OOPSLA*. 503–522.

[10] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB* 12, 11 (2019), 1553–1567.

[11] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

[12] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *PVLDB* 11, 9 (2018), 1002–1015.

[13] Mark Raasveldt and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *Proceedings of Intl. Conference on Scientific and Statistical Database Management*. 16:1–16:12.

[14] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB* 11, 4 (2017), 432–444.

[15] Frank Tip. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3, 3 (1995).

[16] Transaction Processing Performance Council. 2017. TPC Benchmark H. (2017).

Short Paper

Robust and Memory-Efficient Database Fragment Allocation for Large and Uncertain Database Workloads

Rainer Schlosser*
Hasso Plattner Institute, Potsdam, Germany
rainer.schlosser@hpi.de

Stefan Halfpap*
Hasso Plattner Institute, Potsdam, Germany
stefan.halfpap@hpi.de

ABSTRACT

Database replication and query load-balancing are mechanisms to scale query throughput. The analysis of workloads allows load-balancing queries to replica nodes according to their accessed data. As a result, replica nodes must only store and synchronize subsets of the data. However, balancing the load of large-scale workloads evenly while minimizing the memory footprint is complex and challenging. State-of-the-art allocation approaches are either time consuming or the resulting allocations are not memory-efficient. Further, partial replication approaches usually optimize only against a single fixed workload. If the actual workload deviates from this expected one, load-balancing can be highly skewed, resulting in severe performance degradation.

This paper proposes a novel approach to compute memory-efficient fragment allocations that enable balancing multiple potential future workloads. Applied on the TPS-DS benchmark and a large-size enterprise workload, we show that, compared to state-of-the-art allocations, our solutions are (i) more flexible, (ii) require up to 50% less data, (iii) have competitive runtimes, and (iv) are more robust against uncertain out-of-sample workloads.

1 INTRODUCTION

Increasing demand for database processing capabilities can be managed by scale-out approaches, using additional servers. Analyses of enterprise workloads have shown that both OLTP and OLAP are read-dominant [9]. Database replication is an approach to scale-out read-only queries, which can be executed on snapshots of a primary server without violating consistency [4].

Given that most queries require only a limited set of tuples and attributes, partial replication is an efficient approach: Instead of duplicating all data to all replica nodes, partial replicas store only a subset of the data while being able to process a large workload share. Thereby, splitting the overall workload evenly among the replica nodes is essential to scale the query throughput linearly with the number of nodes. Partial replication consists of two steps, which are typically separated from each other to better deal with the problem complexity [11]. First, the data set is partitioned horizontally and/or vertically into disjoint data partitions/fragments. Second, the individual fragments are allocated to one or multiple nodes.

This paper addresses the second step, i.e., a fragment allocation problem (given a fixed data partitioning). Calculating efficient fragment allocations that minimize the replicas' memory consumption while evenly balancing the query load is challenging, because the data allocation and the workload distribution are mutually dependent. As the calculation of optimal allocations is an NP-hard problem, heuristic approaches have to be used for large problem sizes. Rabl and Jacobsen propose a greedy allocation approach with short computation times [12]. We have previously proposed a decomposition approach [5] based on linear programming (LP), which calculates allocations with up to 23% lower memory consumption for the TPC-H benchmark.

Allocations for larger workloads are harder to solve, but typically offer greater potential for sophisticated approaches compared to simple heuristics. However, when using the LP-based decomposition approach for larger problems, computation times increase, and problems may finally become practically intractable, e.g., for an application in dynamic settings, in which model inputs change and quick recalculations are required. Considering multiple potential workloads to increase an allocation's robustness increases the problem complexity even further. However, such robustness is necessary in practice, when workloads fluctuate, and query costs or frequencies cannot be predicted precisely.

The goal of this paper is to overcome the limitations of existing allocation approaches. Applied to TPC-DS and a real-world accounting workload, we show that the greedy rule-based heuristic [12] is not memory-efficient, while the solver-based solution [5] provides low robustness against diversified workloads and has unacceptable runtimes to be used in practice. To fill this gap, we propose a heuristic LP-based clustering approach to *flexibly combine robustness, memory-efficiency, and a short calculation time* for large-scale problems. Our *contributions* are the following:

First, we derive robust and memory-efficient fragment allocations, which enable an even load balancing against multiple potential future workloads. Second, exploiting the skewness of workloads, we use partial clustering techniques to compute solutions for large-scale workloads quickly. Third, for the TPC-DS and a large real-world workload, we show that, with our techniques, the trade-off between memory-efficiency, robustness, and a short calculation time can be smoothly balanced in a targeted way. Fourth, we verify the robustness of our allocations by confronting them with unseen out-of-sample workloads.

2 FRAGMENT ALLOCATION PROBLEM AND LIMITS OF EXISTING APPROACHES

2.1 Problem Description and Difficulty

The scale-out of workloads to partially replicated databases leads to a coupled data assignment and workload distribution problem. We consider a horizontally and/or vertically partitioned database consisting of N disjoint fragments.

The workload input can be described as follows. We consider the case where data (fragments) can be stored on K nodes to distribute a given workload. The size of a fragment i is a_i, $i = 1, ..., N$. Further, we consider a set of Q queries j, characterized by fragments accessed, i.e., $q_j \subseteq \{1, ..., N\}$, $j = 1, ..., Q$. We assume that the costs of queries j are independent of the executing node k, $k = 1, ..., K$, and determined by c_j, $j = 1, ..., Q$. Query costs are numerical and can, e.g., be modeled as average processing times. We assume that the queries j occur with a given frequency f_j, $j = 1, ..., Q$. The total workload costs C are $C := \sum_{j=1,...,Q} f_j \cdot c_j$.

*Both authors contributed equally to this research.

The allocation problem can be described as follows. The goal is to decide (i) on which node to put which fragments and (ii) which query is executed at which node to which extent (workload share). Our objective is to *minimize* the total amount of data at all nodes such that the workload can be evenly distributed between them. Further: (i) A query j can only be executed at node k if all relevant fragments are stored on node k. (ii) For each of the Q queries, the workload shares, assigned to the different nodes, have to sum up to one. (iii) At each of the K nodes, the workload share has to be $1/K$ such that the overall query throughput can be scaled. (iv) Further, as high calculation times may limit the applicability in practice, we want to compute optimized allocations quickly. (v) The allocation should work for multiple given workload scenarios and should be robust against new unseen ones.

2.2 Existing Fragment Allocation Approaches

2.2.1 Optimal Solution via Linear Programming. For a single given workload, the described fragment allocation problem can be formulated as a linear mixed integer problem, cf. [5, 12]. However, the complexity of the linear program quickly increases with the number of queries (Q), fragments (N), and nodes (K). For this reason, the optimal solution can only be derived as long as the size of the problem is sufficiently *small*.

2.2.2 Greedy Heuristic. Rabl and Jacobsen [12] start to assign queries with the largest workload share and accessing the most data. Queries are ordered by the product of the workload share and the total size of accessed fragments. A query is assigned to the node with the largest overlap of already allocated fragments and those accessed by the query. Nodes with no assigned queries are thereby treated as if they have a complete overlap. If a query's workload share exceeds the assigned node's load capacity, the node is filled up to its limit. The query with its remaining workload is merged back into the list of queries and assigned later.

2.2.3 LP-Based Decomposition. We proposed to iteratively *split* the workload into smaller workload packages (chunks) such that the data redundancy is minimized in each step [5]. In a split with B subnodes, each subnode b represents n_b final nodes, $b = 1, ..., B$, and takes the workload share $w_b := n_b/K$. The special case $B = K$ corresponds to the optimal solution, cf. Section 2.2.1. The workload splits are obtained using small-sized LP *subproblems* similar to the LP structure of optimal solutions. In the LP, the following variables are used: The binary variables $x_{i,k} \in \{0, 1\}$, $i = 1, ..., N$, $k = 1, ..., K$, indicate whether fragment i is allocated to node k (1) or not (0). The binary variables $y_{j,k} \in \{0, 1\}$, $j = 1, ..., Q$, $k = 1, ..., K$, indicate whether query j can run on node k (1) or not (0). The continuous variables $z_{j,k} \in [0, 1]$, $j = 1, ..., Q$, $k = 1, ..., K$, represent the workload share of query j executed at node k. The sum of shares has to sum up to one for all queries. Further, by W/V, the *replication factor* is denoted, where the total amount of data used

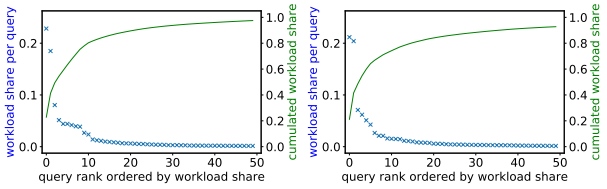
$$W := \sum_{i=1,..,N,k=1,...,K} x_{i,k} \cdot a_i \tag{1}$$

is normalized by the amount of overall accessed data

$$V := \sum_{i \in \bigcup_{j=1,...,Q: f_j > 0} \{q_j\}} a_i. \tag{2}$$

2.3 Large and Skewed Workloads

The results of [5] and [12] were shown for the TPC-H benchmark, which consists of $Q = 22$ queries and $N = 61$ fragments/columns. We use the more complex TPC-DS benchmark ($Q = 99$, $N = 425$) and a real-world enterprise workload ($Q = 4\,461$, $N = 344$).



(a) TPC-DS **(b) Accounting workload**

Figure 1: Distribution of top 50 query workload shares in decreasing order, cf. (a) Section 2.3.1 and (b) Section 2.3.2.

2.3.1 TPC-DS Workload. To obtain model inputs, we loaded the tables with scale factor 1 into a PostgreSQL 12.2 database system. We use vertical partitioning with each column as an individual fragment. We deployed single column indices on all primary key columns. Fragment sizes a_i, $i = 1, ..., 425$, are modeled by using the function `pg_column_size()` to calculate the pure value sizes, abstracting from the PostgreSQL page layout with meta-information and padding. In case the column is part of a primary key, the index size increases the associated fragment size. We use the command `pg_table_size(index_name)` to calculate index sizes. We derived query costs c_j as average execution time for query template j with varying parameters. For TPC-DS queries 1, 4, 6, 11, and 74, the set timeout of 120 s was exceeded. Thus, we omitted them in our experiments, resulting in $Q = 94$ queries.

2.3.2 Real-World Accounting Workload. We got access to metadata of an enterprise's central accounting table and a summary of a workload trace against this table in the form of query templates and statistics. The metadata enabled us to derive all required model inputs for calculating fragment allocations using vertical partitioning. The anonymized workload metadata and source code to reproduce the allocations are publicly available online [1]. The analyzed table stores accounting information and has N=344 columns. The summary of the workload trace consists of Q=4 461 SQL templates with aggregated execution properties of individual queries. Thereby, the most important properties for our research are query frequencies f_j (occurrences) and costs c_j, i.e., the average execution time per query (template).

2.3.3 Workload Skewness. Figure 1 shows the distribution of and cumulative query workload shares $f_j \cdot c_j$, $j = 1, ..., Q$. For both workloads, the distribution is *highly skewed*: The queries with the 50 highest workload shares account for more than 97% of the TPC-DS and more than 92% of the real-world workload.

2.4 Limitations of Existing Approaches

To study the suitability of existing approaches, we calculated allocations for TPC-DS and a real-world workload, cf. Section 2.3. For different numbers K, Table 1 summarizes memory consumption and runtime of existing allocations approaches. We used $f_j := 1$, for all $j = 1, ..., Q$. The LP-results of [5] were achieved using the Gurobi solver (version 9.0.0) (single-threaded). For the comparison with [12]'s approach, we implemented their algorithms in Python 3, and declare the runtime as an upper limit (<).

The upper part of both subtables shows the memory consumption (W/V) and required runtime for the optimal solution, cf. W^D without chunking, for up to $K = 6$ (TPC-DS) and $K = 5$ (accounting workload) nodes compared to the greedy heuristic, cf. W^G. We could not compute optimal allocations for larger numbers of nodes K within 8 hours. The optimal replication factors provide a useful reference to verify the quality of heuristic approaches. In this context, we observe that the greedy heuristic requires up to 100% more data than the optimal solution.

Table 1: Advantages and disadvantages of existing approaches: the greedy heuristic [12] (W^G), cf. Sec. 2.2.2, vs. the decomposition heuristic [5] (W^D), cf. Sec. 2.2.3, including optimal solutions, cf. Sec. 2.2.1, (*no decomposition).

(a) TPC-DS; $K = 2, ..., 12$, $N = 425$, $Q = 94$.

K	chunks	$\frac{W^D}{V}$	solve time$_{W^D}$	$\frac{W^G}{W^D}$	solve time$_{W^G}$
2	2	1.126*	1 s	+1%	<0.1 s
3	3	1.205*	9 s	+53%	<0.1 s
4	4	1.298*	43 s	+94%	<0.1 s
5	5	1.393*	407 s	+88%	<0.1 s
6	6	1.457*	1074 s	+100%	<0.1 s
4	2+2	1.310	2 s	+93%	<0.1 s
5	3+2	1.466	14 s	+79%	<0.1 s
6	3+3	1.519	14 s	+92%	<0.1 s
8	4+4	1.874	122 s	+65%	<0.1 s
10	5+5	2.076	517 s	+61%	<0.1 s
12	6+6	2.201	2 146 s	+60%	<0.1 s

(b) Real-world workload; $K = 2, ..., 12$, $N = 344$, $Q = 4\,461$, cf. source [1].

K	chunks	$\frac{W^D}{V}$	solve time$_{W^D}$	$\frac{W^G}{W^D}$	solve time$_{W^G}$
2	2	1.322*	179 s	+51%	<3 s
3	3	1.775*	6 236 s	+50%	<3 s
4	4	2.104*	9 356 s	+74%	<3 s
5	5	2.473*	13 738 s	+57%	<3 s
3	2+1	1.811	384 s	+47%	<3 s
4	2+2	2.126	225 s	+72%	<3 s
5	2+2+1	2.499	5 922 s	+55%	<3 s
6	3+3	2.855	778 s	+59%	<3 s
8	3+3+2	3.499	8 859 s	+87%	<3 s
10	4+3+3	4.462	49 233 s	+74%	<3 s
12	4+4+4	5.162	47 207 s	+82%	<3 s

The lower part of the subtables shows the results of the decomposition heuristic compared to the greedy heuristic. The decomposition and greedy approach make it possible to solve the problem for larger K heuristically. We observe that the decomposition approach (W^D) yields better replication factors than the greedy approach (W^G), which requires up to 93% and 87% more data for TPC-DS and the accounting workload, respectively. Further, the decomposition approach performs close to optimal compared to the optimal solution results (for $K \leq 6$, see Table 1a and for $K \leq 5$, see Table 1b). However, the decomposition approach requires high computation times if the problem becomes large, e.g., in the case of the accounting workload. In contrast, the greedy approach is fast and requires only seconds.

2.5 Uncertain Future Workloads

In general, future workloads are not entirely predictable. Thus, a further weakness of existing approaches is that they are only optimized for a single workload and can perform poorly if actual workloads differ. Hence, it is crucial to take potential workload scenarios into account to obtain a robust performance. Potential future workload scenarios can be determined, e.g., based on previously observed (seasonal) workloads or forecasts. In this context, fragment allocations should be such that the workload can be successfully balanced in any scenario. We assume that a workload scenario is characterized by a set of queries with given frequencies and costs within a certain time span.

In [12], Rabl and Jacobsen also describe an extension of their approach to cope with multiple workload scenarios. They propose to calculate a separate allocation for each scenario independently. Individual allocations are merged pairwise, mapping each node of the first allocation to a node of the second allocation. A merged allocation enables an even load balancing for both input allocations. The Hungarian method allows calculating an optimal mapping, which minimizes the memory consumption of the merged allocation (in polynomial time). However, because entire nodes are merged, optimization potential is given away.

Overall, we observe that existing approaches have different strengths and weaknesses (runtime vs. memory-efficiency). Further, from a practical perspective, solutions are required that can provide a *reasonable combination* of (i) memory-efficiency, (ii) robustness against different workloads, and (iii) short runtimes. Our goal is to design a heuristic approach that is of that kind.

3 ROBUST FRAGMENT ALLOCATION FOR MULTIPLE POTENTIAL WORKLOADS

3.1 LP-Based Robust Solution Approach

In the following, we consider S potential workload scenarios. Each scenario s, $s = 1, ..., S$, is characterized by query frequencies $f_{j,s}$ and associated workload costs $C_s := \sum_{j=1,...,Q} f_{j,s} \cdot c_j$, cp. Section 2.1. Note, in this framework, also uncertain query costs c_j can be expressed similarly by using potential scenario-based costs $c_{j,s}$ without increasing the model's complexity.

The core idea is finding a single allocation that enables an even load balancing for all S potential workloads scenarios. Further, by enabling an even load balance for specific diversified scenarios, such enriched allocations also allow improved load balancing for *unseen* scenarios, which may be similar to mixtures of input scenarios. Thereby, an allocation's robustness can be increased by choosing a larger number of diverse scenarios.

Our robust multi-scenario model is an extension of the LP-based decomposition approach, cf. Section 2.2.3, which considers one deterministic workload, cf. $S = 1$. Compared to [5], we use extended variables (cf. z), to model workload shares for each scenario s, $s = 1, ..., S$. Based on the decomposition concept of [5], we propose the following extended LP model to allocate data fragments and to distribute workload shares to multiple nodes in the presence of multiple potential workloads:

$$\begin{aligned} &\textit{minimize} \\ &x_{i,b}, y_{j,b} \in \{0,1\}, z_{j,b,s} \in [0,1], 0 \leq L \leq 1, (\text{given } \bar{x}_i, \bar{y}_j, \bar{z}_{j,s}) \\ &i = 1, ..., N,\ j = 1, ..., Q,\ b = 1, ..., B,\ s = 1, ..., S \end{aligned}$$

$$\frac{1}{V} \cdot \sum_{i=1,..,N, b=1,...,B:\bar{x}_i=1} x_{i,b} \cdot a_i + \alpha \cdot L \tag{3}$$

$$s.t. \quad y_{j,b} \cdot |q_j| \leq \sum_{i \in q_j} x_{i,b}, \qquad \begin{matrix} j = 1, ..., Q : \bar{y}_j = 1 \\ b = 1, ..., B \end{matrix} \tag{4}$$

$$z_{j,b,s} \leq y_{j,b}, \qquad \begin{matrix} j = 1, ..., Q : \bar{y}_j = 1 \\ b = 1, ..., B, \quad s = 1, ..., S \end{matrix} \tag{5}$$

$$\sum_{j=1,...,Q:\bar{y}_j=1} f_{j,s} \cdot c_j / C_s / w_b \cdot z_{j,b,s} \leq L, \qquad \begin{matrix} b = 1, ..., B \\ s = 1, ..., S \end{matrix} \tag{6}$$

$$\sum_{b=1,...,B} z_{j,b,s} = \bar{z}_{j,s}, \qquad \begin{matrix} j = 1, ..., Q : \bar{y}_j = 1 \\ s = 1, ..., S \end{matrix} \tag{7}$$

The overall idea is that the model allocates the fragments such that the workload can be distributed evenly for all workload scenarios s, $s = 1, ..., S$. Note, scenario probabilities are not used and thus do not have to be quantified in advance.

The objective (3) minimizes the replication factor W/V, cp. (1) - (2). Constraint (4) guarantees that a query j can only be executed on node b if all relevant fragments are available, see Section 2.1. The cardinality term $|q_j|$ expresses the number of fragments used in query j. Constraint (5) ensures that a query j can only have a positive workload share on node b in scenario s if it can be executed on node b. Constraint (6) guarantees that,

in all scenarios s, all nodes b do not exceed the workload limit L. Here the workload is normalized by the total workload cost C_s of scenario s and the workload share $w_b := n_b/K$, cf. Sec. 2.2.3. Finally, (7) ensures that a query's workload shares on nodes k sum up to the shares assigned to the chunk, cf. $\bar{z}$. To minimize the worst-case workload share over all nodes and scenarios, in (6) we use a continuous variable L and add a penalty term $\alpha \cdot L$ in the objective (3). Hence, to achieve an even load balance, the parameter α has to be sufficiently large compared to K (e.g., $\alpha = 1\,000$); note, the factor W/V is bounded by K, cf. full replication.

The recursive decomposition principle of the LP (3) - (7) works as follows. Let x^*, y^*, and z^* denote the optimal solution of the LP (3) - (7) for a certain split. Then, for each subnode b, the input for its associated subproblem on the next level is characterized by the selected fragments (i.e., where $\bar{x}_{\bar{i}} := x^*_{i,b}$ is 1), the executable queries (i.e., where $\bar{y}_{\bar{j}} := y^*_{j,b}$ is 1), and the assigned workload shares (i.e., where $\bar{z}_{\bar{j},s} := z^*_{j,b,s}$ is positive). In this context, the top node represents the total workload, initially characterized by $\bar{x}_i := 1$, $\bar{y}_j := 1$, $\bar{z}_j := 1$, for all $i = 1, ..., N$ and $j = 1, ..., Q$. On the next decomposition level, the LP (3) - (7) is applied again for the new input. Recall, that for $B = K$ and $w_b = 1/K$, i.e., without using chunks, the LP guarantees an *optimal* allocation.

3.2 Heuristic Relaxation: Partial Clustering

The complexity of the LP (3) - (7) grows with the number of queries Q, fragments N, nodes K, and considered scenarios S. As a result, runtimes can get too large when considering huge workloads with thousands of queries and dozens of scenarios. While the decomposition heuristic allows dealing with larger numbers K, this does not solve the issue. As S can be chosen in a targeted way, the main limitation of our LP-based concept are large Q and N as they appear in real-world workloads (cf. $Q = 4\,461$, Section 2.3.2). Particularly Q is critical for the LP's complexity as the variables y and z as well as constraints (4), (5), and (7) are involved; instead, N is only relevant for x and does not affect the number of constraints. To still allow for short runtimes, we seek to address this problem by heuristically relaxing our LP.

The analysis of real-world workloads reveals, cf. Section 2.3, that the workload distribution of queries and accessed fragments is highly skewed (see Figure 1). Exploiting this property, we cluster queries and, in turn, simplify the complexity of the allocation problem as follows: (i) The majority of queries that represent only a small share of the workload are clustered within a set denoted by Q^F and assigned to the same node. (ii) The set of remaining (costly) queries denoted by, cf. (i),

$$Q^R := \{1, ..., Q\} \setminus Q^F \tag{8}$$

are used as (a smaller) input for the fragment allocation problem with K nodes, including the replica used for step (i). Following this concept, we propose the following approach.

Partial Clustering Approach: *Order the queries according to their expected loads over all scenarios $s = 1, ..., S$, cf. $E(f_{j,s}) \cdot c_j$ (if no distribution for s is given, we use uniform probabilities).* Then, assign the F queries with the smallest (expected) workload share, i.e., $Q^F := \{1, ..., F\}$, to one (e.g., the first) of the K nodes. Note, the number F has to be sufficiently small such that the workload share of the queries assigned to Q^F is (significantly) smaller than $1/K$. *The remaining workload $Q^R := \{F + 1, ..., Q\}$, cf. (8), has to be allocated to the other $K - 1$ nodes and the residual resources of the first cluster node ($k = 1$). The approach can be directly included in our LP model (3) - (7) via the additional constraints*

$$z_{j,1} = 1 \quad \forall j \in Q^F. \tag{9}$$

Note, the constraints (9) imply $y_{j,1} = 1$ for all $j \in Q^F$, cf. (5), as well as the allocation of all required fragments to the cluster node 1, cf. (4). If chunks are used, (9) is only active for the parent chunk $b = 1$ that is associated to the leaf node $k = 1$. Leaving some space on the cluster node allows the LP to assign other data-intensive queries to that node. Naturally, the LP's complexity decreases in F as we have fewer flexible queries ($Q - F$).

4 NUMERICAL EVALUATION

4.1 Results for a Single Fixed Workload

In this section, we compare the memory consumption and runtime of our partial clustering heuristic, cf. (3) - (9), against existing allocation approaches (see Section 2.2) for a single fixed workload scenario ($S = 1$) with $f_{j,1} := 1$, $j = 1, ..., Q$, for all queries.

4.1.1 TPC-DS Workload. For different numbers of nodes K, Table 2a summarizes the replication factor $\frac{W}{V}$ and runtime of our partial clustering heuristic (3) - (9). We used the penalty factor $\alpha := 1\,000$ and the Gurobi solver (version 9.0.0) (single-threaded).

The partial clustering approach allows reducing runtimes (by F via $|Q^F| = F$ and $|Q^R| = Q - F$) and is compatible with the decomposition approach. The results (Table 2a) show that our approach exploits both heuristics in a mutually supportive way. For instance, while for K=6 (TPC-DS), the optimal solution yields $W/V = 1.457$ in 1074 s, cp. Table 1a, our heuristic solution obtains $W/V = 1.584$ in 6 s. Compared to [12] (W^G) and [5] (W^D) we observe that via F we obtain a convenient mix of memory-efficiency and runtime, which has not been possible before.

4.1.2 Real-World Accounting Workload. We also applied our partial clustering approach for the real-world workload, cf. Section 2.3.2. For the example of $F = 4\,361$ fixed and $Q - F = 100$ flexibly assignable queries (being responsible for about 95% of the workload), Table 2b presents the results of our approach compared to the heuristics [5] and [12]. Compared to Table 1b, our partial clustering heuristic yields a competitive memory-efficiency (cf. $W/W^D < +7\%$) and runtimes below 10 s. Overall, we find that our approach can address both the performance limitations of [12] and the runtime limitations of [5].

Table 2: Best of both worlds: Memory-efficiency vs. runtime results achieved by partial clustering ($S = 1$): Our solution (W) with F fixed queries vs. [5] (W^D) and [12] (W^G).

(a) TPC-DS; $K = 4, ..., 12$, $N = 425$, $Q = 94$

K	F	chunks	$\frac{W}{V}$	solve time$_W$	$\frac{W}{W^D}$	$\frac{W}{W^G}$
4	36	4	1.314	1.3 s	+1.2%	-47.9%
5	47	5	1.501	2.0 s	+7.8%	-42.7%
6	4	3+3	1.584	5.5 s	+4.3%	-45.7%
8	15	4+4	1.957	2.6 s	+4.4%	-36.9%
10	47	5+5	2.330	5.0 s	+12.2%	-30.4%
12	15	4+4+4	2.416	7.0 s	+9.8%	-31.2%

(b) Real-world workload; $K = 4, ..., 12$, $N = 344$, $Q = 4\,461$, cf. source [1]

K	F	chunks	$\frac{W}{V}$	solve time$_W$	$\frac{W}{W^D}$	$\frac{W}{W^G}$
4	4 361	4	2.124	3.8 s	+ 0.9%	-41.9%
5	4 361	5	2.492	8.9 s	+ 0.8%	-35.8%
6	4 361	3+3	2.942	0.9 s	+ 3.0%	-35.1%
8	4 361	4+4	3.534	2.1 s	+ 1.0%	-45.9%
10	4 361	5+5	4.638	4.0 s	+ 3.9%	-40.2%
12	4 361	6+6	5.226	25.2 s	+ 1.2%	-44.5%
12	4 361	4+4+4	5.489	3.3 s	+ 6.3%	-41.7%

With decreasing F, the memory W decreases (improves), because the allocation gets more flexible. However, the runtime increases due to the larger remaining problem size. We observe that if the number of flexibly assigned queries $|Q^R|$ is large, the performance increase due to additional flexible queries is *diminishing* and does not justify the higher runtimes anymore.

Remark 1 *The partial clustering heuristic, cf. (3) - (9), is effective and flexible as it combines: (i) the reduction of problem complexity with workload knowledge and (ii) the possibility to exploit decomposition techniques (cf. Section 2.2.3). Combining these techniques allows balancing the solution quality and computation time in a targeted way such that memory-efficient solutions can be computed within short and plannable response times. Compared to the greedy heuristic (W^G), we find that the combined LP-based approach (W) requires (up to 48%) less data (within a similar computation time). Compared to the decomposition approach (W^G), we obtain solutions orders of magnitude (up to ×10 000) faster (using only slightly more memory).*

4.2 Multiple Workloads and Robustness

In this section, we compare fragment allocations for multiple scenarios, cf. $S > 1$. For workload scenario $s = 1$, we again let $f_{j,1} := 1$ for all queries j. For all other scenarios, we consider randomly generated query frequencies $f_{j,s}$ via $f_{j,s} :=$ if $U(0,1) < p$ then $1/p \cdot U(0,2)$ else 0, i.e., each query occurs only with probability p (cf. workloads with different queries, ad-hoc queries, etc.) and, on average, we have $E(f_{j,s}) = 1$, $j = 1, ..., Q$, $s = 2, ..., S$. In our examples, we use $p = 0.75$. Out-of-sample workloads, cf. $\tilde{s} = 1, ..., \tilde{S}$, for verification purposes are generated in the same way. Scenario-specific input details are available online [1].

4.2.1 TPC-DS Workload. Table 3a shows the results of our combined approach (3) - (9), cf. $W(S)$, for different numbers $F = 0, 15, 47, 62$ of fixed queries and different numbers of seen scenarios S (up to 100). While our approach without fixed queries ($F = 0$) is time-consuming, the partial clustering approach is again effective and allows deriving allocations for dozens of scenarios S quickly. We find that the required amount of data $\frac{W}{V}$ is overall (concave) increasing in S. Our replication factor is still significantly below K (cf. full replication) and (for the same S) far better than those of [12]'s merge approach, cf. $W^G(S)$, Section 2.5.

Moreover, we compared the *robustness* of our approach to the merge approach of [12] by analyzing the allocations' performance for *unseen* workloads. To calculate the worst (highest) workload share over all nodes (denoted by $\tilde{L}$) for a given scenario $\tilde{s}$, we can directly use the LP (3) - (7) without chunking ($B = K$) to evaluate a given fragment allocation $\vec{x}_{fix}$ by fixing the variables $\vec{x} := \vec{x}_{fix}$; (this also determines y, cf. (4)). When solving the LP for a *new* (randomly generated) workload scenario $\tilde{s}$, $\tilde{s} = 1, ..., \tilde{S}$, the remaining variables z and L are then chosen such that L is as small as possible and coincides with $\tilde{L}$. The *average* worst-case workload share over all $\tilde{S}$ unseen scenarios is denoted by $E(\tilde{L})$.

Table 3a shows the results for $\tilde{S} = 100$ unseen randomly generated workload scenarios. If more scenarios are taken into account, cf. S, the robustness improves, while the required amount of data and runtime increase. We observe that the average difference of $E(\tilde{L})$ and $1/K$ over all unseen (out-of-sample) scenarios is (concave) decreasing in the number of seen scenarios S. In our example, considering $S = 10$ (randomly chosen) scenarios were, on average, enough to obtain a fragment allocation that is robust against various unseen workloads by achieving an optimality gap for a node's highest workload share of $E^{(S)}(\tilde{L}) - 1/K \leq 0.0089$,

Table 3: Adding Robustness: Performance comparison with $\tilde{S} = 100$ random unseen workloads for different numbers seen workloads S. Memory vs. average out-of-sample workload limits $\tilde{L}$ of our partial clustering ($W(S)$) compared to the greedy merge approach [12] ($W^G(S)$).

(a) TPC-DS; $K = 8 = 4 + 4$, $N = 425$, $Q = 94$, F fixed queries, cf. (9)

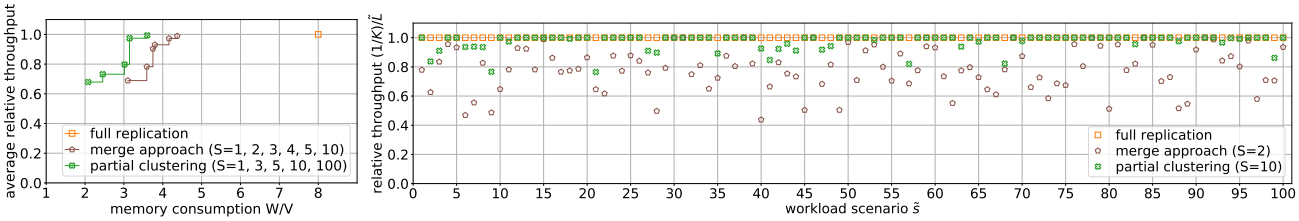
Approach	S	F	$\frac{W}{V}$	solve time	$E(\tilde{L}) - \frac{1}{K}$	$E\left(\frac{1/K}{\tilde{L}}\right)$
$W(S)$	1	0	1.874	110.5 s	0.0641	0.693
$W(S)$	3	0	2.208	175.7 s	0.0538	0.736
$W(S)$	5	0	2.702	345.6 s	0.0305	0.834
$W(S)$	7	0	2.756	286.0 s	0.0342	0.823
$W(S)$	10	0	2.721	561.8 s	0.0073	0.953
$W(S)$	1	47	2.079	1.8 s	0.0681	0.679
$W(S)$	3	47	2.455	2.9 s	0.0537	0.732
$W(S)$	5	47	3.016	2.9 s	0.0388	0.798
$W(S)$	7	47	3.057	6.3 s	0.0325	0.828
$W(S)$	10	15	2.958	42.0 s	0.0089	0.945
$W(S)$	10	47	3.145	20.2 s	0.0037	0.974
$W(S)$	20	47	3.212	11.9 s	0.0037	0.975
$W(S)$	50	47	3.275	36.7 s	0.0015	0.990
$W(S)$	100	47	3.600	331.5 s	0.0010	0.994
$W(S)$	100	62	4.039	112.5 s	0.0005	0.997
$W^G(S)$	1	/	3.101	<1 s	0.0654	0.689
$W^G(S)$	2	/	3.589	<1 s	0.0414	0.782
$W^G(S)$	3	/	3.747	<1 s	0.0155	0.904
$W^G(S)$	5	/	4.162	<1 s	0.0041	0.973
$W^G(S)$	10	/	4.372	<2 s	0.0017	0.989
$W^G(S)$	20	/	4.596	<3 s	0.0005	0.996
$W^G(S)$	50	/	5.528	<6 s	0.0000	1.000

(b) Real-world workload; $K = 8 = 4 + 4$, $N = 344$, $Q = 4\,461$, cf. source [1]

Approach	S	F	$\frac{W}{V}$	solve time	$E(\tilde{L}) - \frac{1}{K}$	$E\left(\frac{1/K}{\tilde{L}}\right)$
$W(S)$	1	4 361	3.534	2.1 s	0.0435	0.769
$W(S)$	3	4 361	3.906	2.8 s	0.0308	0.830
$W(S)$	5	4 361	3.953	6.8 s	0.0264	0.851
$W(S)$	10	4 361	5.008	7.7 s	0.0141	0.921
$W(S)$	10	4 411	5.593	3.8 s	0.0048	0.971
$W(S)$	20	4 361	5.505	8.7 s	0.0007	0.995
$W(S)$	50	4 361	5.743	28.4 s	0.0001	0.999
$W(S)$	100	4 361	5.847	93.7 s	0.0001	0.999
$W(S)$	100	4 411	6.183	23.7 s	0.0000	1.000
$W^G(S)$	1	/	6.536	<3 s	0.0040	0.971
$W^G(S)$	3	/	6.792	<10 s	0.0013	0.991
$W^G(S)$	5	/	6.913	<16 s	0.0013	0.991
$W^G(S)$	10	/	7.040	<34 s	0.0000	1.000

which is by far better than the standard $S = 1$ solution (cf. $E^{(S)}(\tilde{L}) - 1/K = 0.0641$ for $F = 0$) that optimizes only against the expected load. Naturally, the higher robustness with $S = 10$ is achieved by using more data, i.e., a higher replication factor of 2.721 (in 562 s, $F = 0$) and 3.145 (in 20 s, $F = 47$), instead of 1.874 ($F = 0$) and 2.079 ($F = 47$) when using only one scenario ($S = 1$). Note, compared to achieving robustness via full replication (with $W/V = K = 8$), this is a remarkable result.

Further, we evaluated the robustness of the merge approach, cf. $W^G(S)$, for the same ($\tilde{S}$) unseen workloads. For $S = 5$, we obtained the average worst-case workload limit $E^{(G)}(\tilde{L}) - 1/K = 0.0041$ and replication factor $W^G(S)/V = 4.162$. Compared to that, our $S = 10$ solution with $F = 47$ fixed queries obtains a *better robustness* (cf. $E^{(S)}(\tilde{L}) - 1/K = 0.0037$) and requires *clearly less memory* $W^G(S)/V = 3.145$. The better combinations of memory (W/V) and robustness can be observed over the full range of S. Figure 2a visualizes this result: For selected S, the figure shows the expected throughput (average over $\tilde{S} = 100$ unseen workloads)



(a) Memory consumption vs. average relative throughput over 100 unseen scenarios.

(b) Relative throughput for all 100 unseen scenarios.

Figure 2: Performance of allocation approaches for $\tilde{S} = 100$ unseen workload scenarios based on TPC-DS; $K = 8$.

per memory consumption for full replication, the merge approach, and our partial clustering. For merged allocations with $S = 2$ and our partial clustering with $S = 10$ input scenarios, Figure 2b shows the expected throughput $(1/K)/\tilde{L}$ for all $\tilde{S} = 100$ individual unseen scenarios.

Naturally, the specific results depend on underlying random numbers. However, using repeated simulations, we verified that the overall properties remain.

4.2.2 Real-World Accounting Workload. The results for the real-world workload (see Table 3b) show that our solution also remains applicable for larger workloads. The number of fixed queries F can be chosen such that for up to 20 scenarios, the runtime is below 10 s. A good trade-off (depending on the targets in practice) between robustness, runtime, and memory is achieved for 5 to 20 in-sample scenarios, cf. S.

Compared to the merge approach, cf. $W^G(S)$, we again obtain that our approach clearly outperforms their combinations of memory and throughput robustness against uncertain workloads. Moreover, our approach provides the flexibility to *tune* results by adjusting S and F according to a decision-maker's preferences.

Compared to TPC-DS, we find that, for all S, the optimality gap $E^{(S)}(\tilde{L}) - 1/K$ is on average lower, while replication factors are higher. This difference indicates that, in such cases, a *smaller* number of scenarios might be necessary to obtain a certain robustness. It can be explained by the fact that the workload is distributed over a higher number of queries Q, making the impact of single query frequencies, on average, less important.

Remark 2 *We find that optimizing the memory for an expected workload only is not robust against unseen workloads. Less memory-efficient approaches as [12] and particularly its merge extension are (indirectly) more robust against out-of-sample workloads as they systematically allocate more data. However, our approach to directly minimize required memory for multiple seen workloads, yields a better robustness using the same or even less data. The memory-efficiency of our LP-based approach allows including more scenarios within a certain memory budget than the merge approach [12]. Being able to deal with more representative workload scenarios, in turn, allows to better deal with altered unseen ones.*

5 RELATED WORK

Database replication is a means to improve availability and increase processing capabilities, and is supported in many systems, e.g., in HANA [10], Postgres-R [8], and as replication middleware [3]. Thereby, most systems implement full replication.

To calculate partial allocations, Rabl and Jacobsen propose a greedy algorithm (Section 2.2.2). For the same problem, we propose an LP-based decomposition approach [5] (Section 2.2.3). In both papers, the authors only consider a comparably small benchmark workload (TPC-H) and do neither (i) derive results for large-size workloads, (ii) study techniques to reduce the computation time for allocations, nor (iii) evaluate robustness against uncertain workloads. In [6], we visualize calculated allocations and investigate the intermediate steps taken by different algorithms. In [7], we visualize the load balancing behavior of allocations.

Özsu and Valduriez present a general overview of allocation problems in the field of distributed database systems [11]. They point out that allocation problems differ in constraints and optimization goals, such as performance, costs, and dependability.

Archer et al. [2] address an allocation problem, which is similar to ours. They evenly load-balance queries for web search containing multiple terms, which correspond to the fragments in our model. They cluster queries with a distributed balanced graph partitioning tool.

6 CONCLUSIONS

To overcome the limitations of existing allocation approaches [5, 12], we proposed a novel heuristic that combines different concepts. First, to achieve *memory-efficiency*, instead of rule-based heuristics, we leverage the power of LP techniques. Second, for *short calculation times*, we exploit that (real-world) workloads are skewed and reduce the problem complexity by a partial clustering approach that assigns a majority of queries with comparably small workload shares to a fixed node. Third, to add *robustness*, we force the allocation to be prepared against multiple diversified workloads. Moreover, we are able to balance the three target dimensions of the problem *flexibly*. Using TPC-DS and a large real-world workload, we verified the applicability and effectiveness of our approach, which clearly outperformed existing approaches regarding the mix of the three key dimensions.

REFERENCES

[1] [n.d.]. GitHub repository containing the workload data and source code. https://hyrise.github.io/replication/.
[2] Aaron Archer et al. 2019. Cache-aware load balancing of data center applications. *PVLDB* 12, 6 (2019), 709–723.
[3] Emmanuel Cecchet et al. 2008. Middleware-based database replication: the gaps between theory and practice. In *SIGMOD*. 739–752.
[4] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. 1996. The Dangers of Replication and a Solution. In *SIGMOD*. 173–182.
[5] Stefan Halfpap et al. 2019. Workload-Driven Fragment Allocation for Partially Replicated Databases Using Linear Programming. In *ICDE*. 1746–1749.
[6] Stefan Halfpap and Rainer Schlosser. 2019. A Comparison of Allocation Algorithms for Partially Replicated Databases. In *ICDE*. 2008–2011.
[7] Stefan Halfpap and Rainer Schlosser. 2020. Exploration of Dynamic Query-Based Load Balancing for Partially Replicated Database Systems with Node Failures. In *CIKM*. 3409–3412.
[8] Bettina Kemme et al. 2000. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *VLDB*. 134–143.
[9] Jens Krüger et al. 2011. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB* 5, 1 (2011), 61–72.
[10] Juchang Lee et al. 2017. Parallel Replication across Formats in SAP HANA for Scaling Out Mixed OLTP/OLAP Workloads. *PVLDB* 10, 12 (2017), 1598–1609.
[11] M. Tamer Özsu and Patrick Valduriez. 2011. *Principles of Distributed Database Systems, Third Edition*. Springer.
[12] Tilmann Rabl and Hans-Arno Jacobsen. 2017. Query Centric Partitioning and Allocation for Partially Replicated Database Systems. In *SIGMOD*. 315–330.

Short Paper

TD-AC: Efficient Data Partitioning based Truth Discovery

Osias Noël Nicodème Finagnon TOSSOU
African Institute for Mathematical Sciences
Mbour, Senegal
osias.tossou@aims-senegal.org

Mouhamadou Lamine Ba
Université Alioune Diop de Bambey
Bambey, Senegal
mouhamadoulamine.ba@uadb.edu.sn

ABSTRACT

This paper introduces an effective algorithm, called **TD-AC**, for the truth discovery problem in scenarios where data attributes are *correlated* by distinct levels of reliability of the sources. **TD-AC** is built on an abstract representation of the truth in the data to automatically find an optimal partitioning of the input data using the k-means clustering technique and the silhouette measure. Such a data partitioning strategy ensures to maximize the accuracy of any *base* truth discovery process when executed on each partition. The intensive experiments conducted on synthetic and real datasets show that **TD-AC** outperforms baseline approaches with a more reasonable running time. It improves on synthetic datasets the accuracy of standard truth discovery algorithms by 1% at least and by 14% at most and also significantly when the data coverage rate is high for the other types of datasets.

KEYWORDS

Truth discovery, attribute, data partitioning, clustering, attribute truth vector, k-means, silhouette index, performance evaluation

1 INTRODUCTION

Dealing with contradictory claims about the same facts is a real concern in many real-world applications such as Web data integration systems [3], online crowdsourcing platforms, online news Websites, social media, etc. Truth discovery resolves such a issue by predicting which of the values provided by conflicting sources is true with no prior knowledge about the level of reliability of the sources. Many approaches [1, 2, 5, 7, 12] for truth discovery have been proposed based on an estimation of the reliability of sources by corroborating their claims under various settings. As in [2], we investigate in this work the truth discovery with attribute partitioning problem that may occur in cases where the attributes over data are structurally correlated so that sources exhibit different levels of reliability on distinct groups of data attributes, as in the setting given in Table 1. Table 1 shows conflicting claims about facts (or data attributes) on two distinct topics (Table 1b) from three sources as depicted in Table 1a. Given the correct answers inside red ellipses, we note that the sources present different levels of reliability according to distinct subsets of facts. For instance, Source 1 is good on Q1 and Q3 while being bad on Q2. Meanwhile Source 2 is good on Q2 and bad on Q1 and Q3. We say that Q1 and Q3 are about data attributes that are *correlated* according to the sources' reliability levels ; *capturing these unknown groups of correlated attributes may help to avoid having a biased truth discovery process.*

The approach in [2] finds the set of correlated data attributes for truth discovery as an optimal partitiioning of the set of input data attributes using various weighting functions over sources' reliability levels themselves estimated by the truth discovery

Football (FB)	Q1 - Which country won the 2019 Africa Cup of Nations? Q2 - In which year did Benin reach the quarter-finals for the first time in the Africa Cup of Nations? Q3 - How many players are there per team in a football-game?
Computer science (CS)	Q1- Who created the kernel of the linux system? Q2- In which year did he create it? Q3 - What does this python code display? *print(3+4)*

(a) Several facts about two different topics

Sources	Topic	Q1	Q2	Q3
Source 1	FB	Algeria	2000	12
Source 2	FB	Senegal	2019	11
Source 3	FB	Algeria	1994	12
Source 1	CS	Linux Torvalds	1830	7
Source 2	CS	Bill Gate	1991	8
Source 3	CS	Steve Jobs	1991	10

(b) Source claims about those facts

Table 1: Example with sources having different levels of reliability with respect to distinct groups of data attributes

algorithm. However, the different exploration strategies introduced in [2] are *time-consuming* and *error-prone*. In addition, its different weighting functions do not give any guarantees about the correctness of the returned optimal partition.

This paper revisits [2] and proposes a new more effective and efficient approach to the problem of truth discovery with attribute partitioning. The presented approach, called **TD-AC**, is based on an abstract representation of the truth in the data using the new concept of *attribute truth vector*. Given the set of attribute truth vectors, we rely on *k-means clustering* technique from machine learning domain to find the optimal partitioning of the data attributes. To determine the optimal number of clusters, we assess the homogeneity of the individuals in a clustering result with the help of the silhouette measure. This methodology guarantees to find an optimal partition or a near-optimal one maximizing the accuracy of any base truth discovery process, without an exploration of all the possible partitions. The results of our intensive experiments on synthetic, semi-synthetic and real datasets show that **TD-AC** outperforms approaches in [2], with a more reasonable time cost. On synthetic data, it improves the accuracy of standard algorithms at least by 1% and at most

by 14% and also significantly when the data coverage is high for the other types of datasets.

The remainning of this paper is organized as follow. First, we give some preliminaries and define the studied problem in Section 2. Then, we detail our proposed approach by providing its different building blocks in Section 3. In order to validate our approach, we present in Section 4 the results of our intensive experiments conducted on various types of datasets and a thorough analysis of the obtained results. We briefly review the state-of-the-art truth discovery algorithms in Section 5 before concluding in Section 6 with some research perspectives.

2 CONCEPTS AND STUDIED PROBLEM

This section resumes the key concepts of the truth discovery problem and informally introduces the studied problem.

2.1 DEFINITION OF CONCEPTS

A typical truth discovery process usually assumes a *structured world* where input data consist of a set O of objects corresponding to real world entities. Each object is characterized by a set A of attributes (or properties) with values in V coming from a collection S of data sources. In a *one-truth setting*, every attribute for each object has one true value and several possible false values. Thus, the notion of value confidence C_v is used to assess the level of veracity of every value v. Meanwhile, the level of reliability T_s of a source s (or source accuracy) models its ability to provide true values for given real-world object attributes. In real applications, the confidence scores over provided values and the reliability levels of sources are both often unknown and initialized to default values depending on the setting before being updated during the execution of the truth discovery algorithm. *This work considers groups of attributes over data to be structurally correlated if every source has the same reliability level on these latter.*

2.2 PROBLEM STATEMENT

Given the triplet (S, A, O) in a one-truth setting in which a given source may not cover all the objects or attributes, the truth discovery problem is commonly defined as follows.

Problem 1. *Find, for each object o in O, the true value of every attribute a in A_o amongst its set V_{o-a} of possible values by corroborating claims from sources in S_o where A_o and S_o are the set of attributes of o and the set of sources providinng values for o.*

We informally introduce the truth discovery with attribute partitioning problem as follows.

Problem 2. *Find an optimal partitioning P of A that maximizes the accuracy of any solution for Problem 1 where each partition in P contains correlated data attributes according to sources' reliability levels.*

In next, we propose an efficient clustering based approach to solve Problem 2 when data attributes are structurally correlated.

3 TRUTH DISCOVERY WITH CLUSTERING

This section presents our proposed algorithm, called **TD-AC**, that discovers the truth by data partitioning. **TD-AC**, that stands for *Truth Discovery with Attribute Clustering*, applies *k-means* to find optimal clusters of structurally correlated data attributes based on sources' reliability level by relying on *attribute truth vectors* and the *silhouette index*, as we detail it below.

3.1 DATA ATTRIBUTE TRUTH VECTORS

We define and use the concept of *data attribute truth vectors* as an abstract representation of the precision (or quality) of a given truth discovery algorithm using attributes as dimensions. To build such vectors, we firstly apply a *base truth discovery algorithm* (e.g. majority voting) on input data to obtain a *reference truth*. Then, for each attribute of an object and every source we verify whether or not the value given by the source is true regarding the reference truth; we set the value for each rank of any attribute truth vector according to Equation 1.

$$\forall a \in A, \forall o \in O, \forall s \in S; x(a,o,s) = \begin{cases} 1 & if\, \rho\ is\ true \\ 0 & otherwise \end{cases} \quad (1)$$

where $\rho = (v(a,o,s)\ \text{exists}\ \wedge\ v(a,o,s) = v_F(a,o))$ with $v(a,o,s)$ representing the value given by s about a of o, $v_F(a,o)$ is the true value of a of o predicted by the base algorithm and $x(a,o,s)$ is a binary value of our truth vector. Table 2 sketches the matrix of attribute truth vectors obtained on our running example in Table 1 by applying the procedure described above and Equation 1.

-	FB	CS	FB	CS	FB	CS
Q1	1	0	0	0	1	1
Q2	0	0	1	1	0	1
Q3	1	0	0	0	1	1

Table 2: Matrix of attribute truth vectors with data in Table 1 using TruthFinder as base algorithm

3.2 GROUPING CORRELATED ATTRIBUTES

We find and group correlated data attributes by assessing the similarity distance of their corresponding truth vectors. Given two distinct attributes a_1, a_2 and their truth vectors $(a_1^1, a_1^2, \ldots, a_1^l)$ and $(a_2^1, a_2^2, \ldots, a_2^l)$, we define the similarity between a_1 and a_2 using the Hamming distance as : $d(a_1, a_2) = \sum_{i=1}^{l} |a_1^i - a_2^i|$ (2).

To automatically devise the *threshold value* for grouping the attributes based on our similarity measure, we rely on *k-means* and its optimization strategy in order to provide a domain-independent clustering process in practical cases. The k-means clustering approach [8] uses a similarity distance metric between data points to group them in k clusters. Given a set of observations $(a_1, a_2, ..., a_n)$, where every observation is an attribute truth vector having l dimensions, we define the partitioning of these attributes using k-means algorithm as the clustering of the n observations in k ($k \leq n$) disjoint sets (or clusters) $C = \{g_1, g_2, ..., g_k\}$ in such a way that the sum of the squares (i.e. the Inertia) within each cluster is minimized. Formally, the goal is to find:
$argmin_C \sum_i^k \sum_{a \in g_i} ||a - \mu_i||^2 = argmin_C \sum_i^k |g_i| Inertia(g_i)$ (3)
where μ_i is the centroid of the points in g_i. This corresponds to minimize the squared deviations of the points in the same cluster: $argmin_C \sum_i^k \sum_{a_1 \in g_i} \frac{1}{2|g_i|} \sum_{a_1, a_2 \in g_i} ||a_1 - a_2||^2$ (4). K-means requires to specify the value of k in input. We find the optimal k using the *silhouette index* as described next.

3.3 ESTIMATION OF k WITH SILHOUETTE

The silhouette index [11] evaluates the quality of a clustering result with the help of the separation criteria β and the cohesion criteria α. Consider two attributes a_1 and a_2 that belong to clusters $g(1)$ and $g(2)$, respectively. Formally, the silhouette coefficient $CS(a_1)$ of the attribute a_1 is defined as: $CS(a_1) = \frac{\beta(a_1) - \alpha(a_1)}{\max(\alpha(a_1), \beta(a_1))}$ with $\alpha(a_1) = \frac{1}{|g(1)|-1} \sum_{a_j \in g(1); a_j \neq a_1} d(a_1, a_j)$ and $\beta(a_1) = \min_{a_1 \neq a_2} \frac{1}{|g(2)|} \sum_{a_k \in g(2)} d(a_1, a_k)$ (5). If $CS(a_1) <$

0, a_1 is *badly classified*. Conversely, if $CS(a_1) > 0$ a_1 is *well classified*. Finally, if $CS(a_1) = 0$ then a_1 is between two clusters. The silhouette coefficient $CS(g)$ of a cluster g is thus given by: $CS(g) = \frac{1}{|g|} \sum_{a \in g} CS(a)$ (6).

The silhouette value of a partition P is the average of the silhouette coefficients of all its clusters: $CS(P) = \frac{1}{|P|} \sum_{g \in P} CS(g)$ (7).

The optimal k is the one associated to the partition having the highest silhouette coefficient amongst all the possible partitions.

3.4 TD-AC TRUTH DISCOVERY APPROACH

As depicted by Algorithm 1, our proposed algorithm **TD-AC** runs as follows: (i) considers a *base truth discovery algorithm* and input data (A, O, S); (ii) computes the matrix of attribute truth vectors from input data using the base algorithm and Equation 1; (iii) efficiently clusters the data attributes by applying k-means combined with the silhouette index ; and (iv) executes the input base truth discovery algorithm on each data partition, and then aggregates the partial results to generate the entire result.

Algorithm 1 TD-AC(F, A, O, S) – Truth discovery with Attribute clustering using k-means and silhouette coefficient

Require: Set of observations (A, O, S), Base algorithm F
Ensure: *results* // Truth predicted by TD-AC
1: *results* ← [ ]
2: *truth_vector_matrix* ← *buildTruthVectors*(F, A, O, S)
3: // Find the optimal partition with k-mean and silhouette
4: *indice_silhouette* ← 0
5: *opt_partition* ← []
6: **for all** $k \in [2, |A| - 1]$ **do**
7: *partition=kmeansAttClustering*(*truth_vector_matrix*,k)
8: *silhouette_index_tmp* ← *CS(partition)*
9: **if** $k == 2$ **then**
10: *silhouette_index* ← *silhouette_index_tmp*
11: *opt_partition* ← *partition*
12: **else**
13: **if** $indice\_silhouette < indice\_silhouette\_tmp$ **then**
14: *silhouette_index* ← *silhouette_index_tmp*
15: *opt_partition* ← *partition*
16: **end if**
17: **end if**
18: **end for**
19: // Truth discover on the optimal partition found
20: **for each** $g \in opt\_partition$ **do**
21: $A_p, O_p, S_p \leftarrow getData(g)$
22: *partial_result* ← $F(A_g, O_g, S_g)$
23: Add *partial_result* in *results*
24: **end for**

4 EXPERIMENTS AND RESULTS

In this section, we demonstrate the efficiency of our approach on various datasets, proving that it outperforms approaches proposed in [2] and standard truth discovery algorithms in the literature in the presence of structurally correlated data attributes. We also show that its execution time is similar to that of standard algorithms unlike partitioning strategies in [2]. We start by presenting the experiment setting up and performed tests.

4.1 EXPERIMENTATION SETTING UP

For the comparison purposes, we have implemented the different analyzed algorithms using *Python* programming language. The following standard truth discovery algorithms have been implemented: **MajorityVote**,**TruthFinder** [14], **DEPEN**, **Accu** and **AccuSim** [4]. We have compared ourselves to these algorithms because they are amongst the best in terms of efficiency and effectiveness for solving the truth discovery problem in various settings. In addition we have also implemented **AccuGenPartition** in [2] along with the different weighting functions to compute the optimal partition. The source codes of the tested algorithms are all available at **https://github.com/osiastossou/ProjetTD-AC.git**.

	DS1	DS2	DS3
m_1	1.0	1.0	1.0
m_2	0.0	0.0	0.2
m_3	1.0	0.8	0.8

Table 3: Average accuracy values for the various configurations of the synthetic datasets

We have conducted all our experiments on a Intel Core i5 2.6GHz laptop computer with 8GB of RAM, 250GB of hard disk space, and 1.5GB of graphics memory. The implemented algorithms here require all hyper-parameters in input whose values have been fixed for the various tests according to [12]. At last, we have relied on usual metrics such as *precision*, *recall*, *F1-measure*, *accuracy*, and *execution time* to evaluate and compare the performance of our tested algorithms.

4.2 EXPERIMENTS ON SYNTHETIC DATA

We detail here the results of our experiments on synthetic data which simulate conditions where data attributes are structurally correlated.

We have used and re-implemented in Python the synthetic data generator in [2] to produce our synthetic data sets; we defer to [2] for the details. For the evaluation process, we have then generated three synthetic datasets (DS1, DS2 and DS3) of 6 attributes, 1000 objects, 10 sources and 60, 000 observations with three different configurations as depicted in Table 3; DS1 meets the seeting of this work while DS3 relaxes the assumptions to test the robustness of our approach. The partition selected for each configuration is given in Table 5.

Tables 4a, 4b and 4c respectively present the performances of each algorithm on DS1, DS2 and DS3. For the tests, we used **Accu** as our base algorithm similarly to the approaches in [2]. We observe that the attribute partitioning truth discovery algorithms perform better than the standard ones on all three synthetic datasets, proving the importance of partitioning when data attributes are structurally correlated. Specifically, **TD-AC** is the only partitionning strategy with a precision comparable to the real world (i.e. an *Oracle*) without a blowup of the running time. Table 5 reports the partitions returned by the different partitioning approaches.

4.3 TESTS ON SEMI-SYNTHETIC DATA

The semi-synthetic datasets have been generated from a real dataset called **Exam**. This real dataset comes from [2] and has been used in that paper to validate the proposed approaches. The **Exam** dataset has been obtained by aggregating the anonymous results of admission examinations. Unfortunately, it cannot be redistributed for privacy reasons. We had access to answers from 248 students (*sources*) to 124 questions (*attributes*) in total, from 9 different domains: **Math 1A, Chemistry 1, Math 1B, Physics, Electrical Engineering, Computer Science, Chemistry 2, Science of life,** and **Math 2**. We also know the correct answer to each question. Math 1A and Physics were only mandatory with the choice of an additional domain between Chemistry 1 and Math 1B. The five remaining domains were completely optional and wrong answers were penalized. As a result, all the attributes were not covered (missing data). For each unanswered

Dataset	Algorithm		Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
DS1	**MajorityVote**		0.602	0.667	0.806	0.633	75	1
	TruthFinder		0.568	0.624	0.787	0.595	1261	3
	DEPEN		0.551	0.611	0.778	0.580	1492	3
	Accu		0.667	0.712	0.838	0.689	6495	9
	AccuSim		0.662	0.705	0.836	0.683	5580	11
	AccuGenPartition	**Max**	0.691	0.724	0.849	0.707	757230	-
		Avg	0.682	0.725	0.846	0.703	757230	-
		Oracle	0.997	0.998	0.999	0.998	757230	-
	TD-AC (F=Accu)		0.853	0.870	0.930	0.861	3410	1

(a) Performance measures on DS1

Dataset	Algorithm		Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
DS2	**MajorityVote**		0.741	0.834	0.884	0.785	99	1
	TruthFinder		0.736	0.819	0.880	0.775	2276	3
	DEPEN		0.735	0.828	0.881	0.779	1459	3
	Accu		0.659	0.663	0.828	0.661	11263	18
	AccuSim		0.467	0.388	0.734	0.424	9996	20
	AccuGenPartition	**Max**	0.738	0.810	0.879	0.773	861697	-
		Avg	0.867	0.904	0.940	0.885	861697	-
		Oracle	0.985	0.992	0.994	0.989	861697	-
	TD-AC (F=Accu)		0.985	0.992	0.994	0.989	3783	1

(b) Performance measures on DS2

Dataset	Algorithm		Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
DS3	**MajorityVote**		0.847	0.891	0.918	0.869	112	1
	TruthFinder		0.838	0.875	0.910	0.856	2762	3
	DEPEN		0.833	0.876	0.909	0.854	1732	3
	Accu		0.873	0.918	0.934	0.895	3478	7
	AccuSim		0.808	0.822	0.886	0.815	7171	15
	AccuGenPartition	**Max**	0.872	0.884	0.925	0.878	675078	-
		Avg	0.938	0.958	0.968	0.948	675078	-
		Oracle	0.965	0.976	0.982	0.970	675078	-
	TD-AC (F=Accu)		0.965	0.976	0.982	0.970	2491	1

(c) Performance measures on DS3

Table 4: Performance of all tested algorithms on the synthetic datatsets DS1, DS2 and DS3

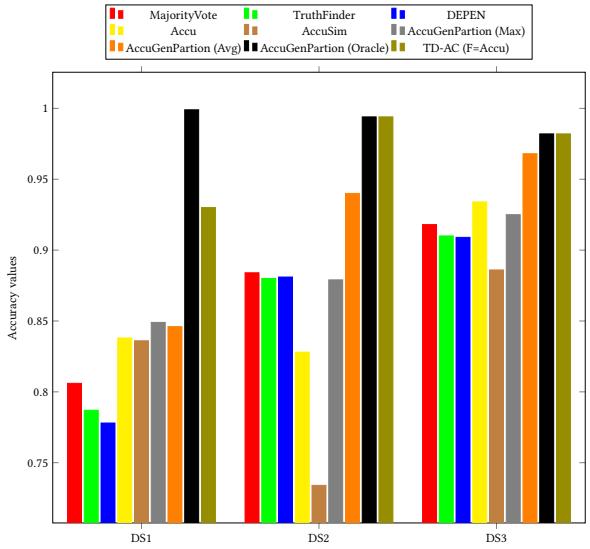


Figure 1: Comparison of the accuracy of all tested algorithms on DS1, DS2 and DS3

	DS1	DS2	DS3
Synthetic data generator	[(1, 2), (4, 6), (3), (5)]	[(2, 5), (1, 4), (3, 6)]	[(1, 6, 3), (2, 4, 5)]
AccuGenPartition (Max)	[(3, 4), (5), (1, 2, 6)]	[(2), (1, 4, 3, 5, 6)]	[(1), (5, 2, 4, 3, 6)]
AccuGenPartition (Avg)	[(3, 6), (1, 2, 5, 6)]	[(2), (5), (1, 4, 3, 6)]	[(1, 5), (2, 4, 3, 6)]
AccuGenPartition (Oracle)	[(1), (2), (3), (4, 6), (5)]	[(2, 5), (1), (4), (3, 6)]	[(1, 5), (2, 4), (3, 6)]
TD-AC (F=Accu)	[(1, 2), (4, 6), (3, 5)]	[(2, 5), (1, 4), (3, 6)]	[(1, 5), (2, 4), (3, 6)]

Table 5: Partitions chosen by the generator and returned by the different partitioning algorithms

question we have synthetically chosen a false answer, randomly in a range of false values of size equal to 25, 50, 100 or 1000.

Tables 6 and 7 respectively present the different results of these tests on the semi-synthetic data of 62 and 124 attributes, each with configurations on ranges of false values of size 25, 50, 100, and 1000. The tests compare the performances of **Accu** and **TD-AC+Accu** on the one hand and on the other hand **TruthFinder** and **TD-AC+TruthFinder**. In general, we note that combining a base algorithm with **TD-AC** does not highly deteriorate the performance of the standard algorithm whatever the configuration considered, and even improves it in some cases, for example for the dataset with 124 attributes (see Figures 2 and 3).

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Range 25	**Accu**	0.929	0.896	0.938	0.912	4386	8
	TD-AC (F=Accu)	0.920	0.883	0.931	0.901	10256	1
	TruthFinder	0.894	0.917	0.931	0.905	85	6
	TD-AC (F=TruthFinder)	0.897	0.920	0.933	0.908	62	1

(a) Range 25

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Range 50	**Accu**	0.946	0.912	0.951	0.928	4615	**8**
	TD-AC (F=Accu)	0.963	0.970	0.976	0.966	18233	1
	TruthFinder	0.915	0.934	0.946	0.924	80	4
	TD-AC (F=TruthFinder)	0.915	0.934	0.946	0.924	81	1

(b) Range 50

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Range 100	**Accu**	0.988	0.983	0.990	0.985	4017	7
	TD-AC (F=Accu)	0.972	0.982	0.984	0.977	7684	1
	TruthFinder	0.924	0.943	0.954	0.933	134	3
	TD-AC (F=TruthFinder)	0.925	0.944	0.955	0.935	121	1

(c) Range 100

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Range 1000	**Accu**	0.989	0.984	0.991	0.986	4186	7
	TD-AC (F=Accu)	0.972	0.982	0.984	0.977	8467	1
	TruthFinder	0.927	0.946	0.956	0.936	258	4
	TD-AC (F=TruthFinder)	0.927	0.946	0.956	0.936	241	1

(d) Range 1000

Table 6: Performance of Accu, TruthFinder, TD-AC(F=Accu), and TD-AC(F=TruthFinder) on semi-synthetic datasets with 62 attributes

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Range 25	**Accu**	0.847	0.739	0.904	0.789	7805	9
	TD-AC (F=Accu)	0.852	0.744	0.906	0.794	12432	1
	TruthFinder	0.894	0.919	0.954	0.906	104	3
	TD-AC (F=TruthFinder)	0.894	0.919	0.954	0.906	157	1

(a) Range 25

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Range 50	**Accu**	0.885	0.806	0.931	0.844	10680	11
	TD-AC (F=Accu)	0.928	0.916	0.964	0.922	10456	1
	TruthFinder	0.906	0.931	0.962	0.918	278	4
	TD-AC (F=TruthFinder)	0.904	0.929	0.961	0.916	276	1

(b) Range 50

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Range 100	**Accu**	0.905	0.822	0.943	0.862	8516	**10**
	TD-AC (F=Accu)	0.953	0.955	0.980	0.954	10196	1
	TruthFinder	0.905	0.918	0.961	0.911	597	5
	TD-AC (F=TruthFinder)	0.909	0.934	0.965	0.921	460	1

(c) Range 100

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Range 1000	**Accu**	0.930	0.913	0.966	0.921	11951	12
	TD-AC (F=Accu)	**0.934**	0.927	**0.970**	**0.931**	9222	1
	TruthFinder	0.921	0.941	0.970	0.931	1626	4
	TD-AC (F=TruthFinder)	0.909	0.933	0.965	0.921	1401	1

(d) Range 1000

Table 7: Performance of Accu, TruthFinder, TD-AC(F=Accu), and TD-AC(F=TruthFinder) on semi-synthetic datasets with 124 attributes

4.4 EXPERIMENTS ON REAL DATA

To end our performance evaluation, we present in this section the results of the experimentation of our approach and the existing algorithms on real data. The evaluation on real data sets enables to validates our approach against practical applications. For this purpose, we have considered and used the real datasets **Exam** [2], **Stocks** and **Flights** [9]. Real data contain missing values that

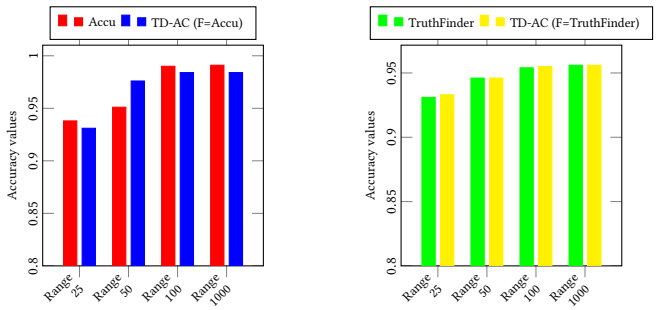


Figure 2: Study of the impact of TD-AC on Accu and TruthFinder by pairwise comparison of the accuracy values on semi-synthetic datasets with 62 attributes

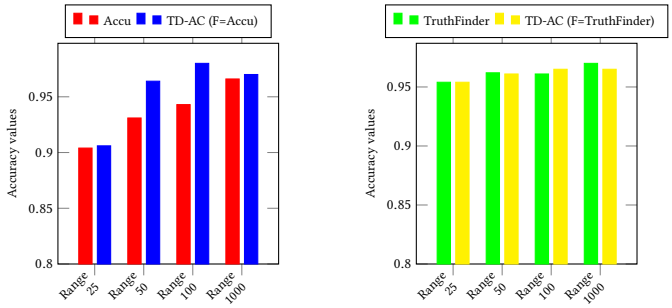


Figure 3: Study of the impact of TD-AC on Accu and TruthFinder by pairwise comparison of the accuracy values on semi-synthetic datasets with 124 attributes

may impact the performance of truth discovery algorithms. We assess the *Data Coverage Rate* (DCR) of each dataset with: $DCR = \left(1 - \frac{\sum_{o \in O}(|S_o| \times |A_o| - \sum_{s \in S_o} |A_{o-s}|)}{\sum_{o \in O}(|S_o| \times |A_o|)}\right) \times 100$ (7). Table 8 presents the details of the three real data sets after pre-processing; for Exam we have considered three configurations.

	Stocks	Exam 32	Exam 62	Exam 124	Flights
Number of sources	55	248	248	248	38
Number of objects	100	1	1	1	100
Number of attributes	15	32	62	124	6
Number of observations	56992	6451	8585	11305	8644
Data Coverage Rate (%)	75	81	55	36	66

Table 8: Statistics about the different real datasets

Table 9 presents the performance measures of **Accu**, **TD-AC+Accu**, **TruthFinder**, and **TD-AC+TruthFinder**. We have also reported in Figures 4 and 5 the comparative study of the accuracy values of **Accu** and **TD-AC+Accu** on the one hand and **TruthFinder** and **TD-AC+TruthFinder** on the other hand on real datasets with data coverage greater than 66% and less than 55% respectively. We observe that **Accu** and **TruthFinder** outperform when used with our **TD-AC** approach, especially when the data coverage rate is greater than 66%. We also remark that the execution time of **TD-AC** is very close to that of standard algorithms on real data, unlike **AccuGenPartition**.

4.5 Analysis of the results and discussion

The analysis of the presented intensive performance evaluation carried out on several datasets yields to three main observations.

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Exam 32	**Accu**	0.607	0.837	0.658	0.704	4059	11
	TD-AC (F=Accu)	0.614	0.912	0.679	0.734	4075	1
	TruthFinder	0.540	0.772	0.570	0.636	6.66	5
	TD-AC (F=TruthFinder)	0.533	0.733	0.558	0.617	13.7	1

(a) Exam with 32 attributes

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Exam 62	**Accu**	0.955	0.962	0.944	0.959	4877	10
	TD-AC (F=Accu)	0.926	0.944	0.911	0.935	2789	1
	TruthFinder	0.937	0.955	0.926	0.945	16.2	5
	TD-AC (F=TruthFinder)	0.898	0.885	0.854	0.891	24.3	1

(b) Exam with 62 attributes

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Exam 124	**Accu**	0.951	0.969	0.947	0.960	3662	9
	TD-AC (F=Accu)	0.917	0.938	0.904	0.927	3733	1
	TruthFinder	0.924	0.949	0.916	0.936	23.5	5
	TD-AC (F=TruthFinder)	0.907	0.906	0.878	0.907	79	1

(c) Exam with 124 attributes

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Stocks	**Accu**	0.847	0.877	0.809	0.862	2753	4
	TD-AC (F=Accu)	0.886	0.956	0.887	0.920	4169	1
	TruthFinder	0.860	0.700	0.718	0.772	629	5
	TD-AC (F=TruthFinder)	0.887	0.862	0.832	0.875	446	1

(d) Stocks

Dataset	Algorithm	Precision	Recall	Accuracy	F1-measure	Time(s)	#Iteration
Flights	**Accu**	0.958	0.968	0.957	0.963	390	7
	TD-AC (F=Accu)	0.969	0.987	0.974	0.978	452	1
	TruthFinder	0.859	0.900	0.857	0.879	22.3	3
	TD-AC (F=TruthFinder)	0.848	0.885	0.842	0.866	33	1

(e) Flights

Table 9: Performance of Accu, TruthFinder, TD-AC+Accu, and TD-AC+TruthFinder on real datasets

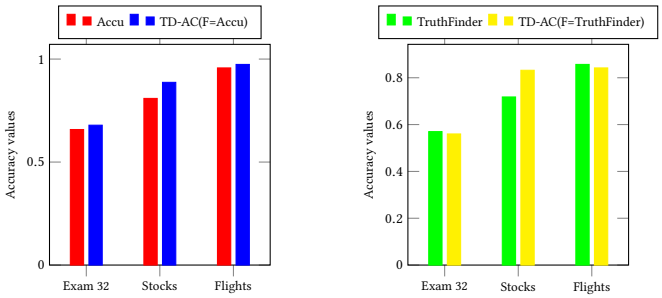


Figure 4: Study of the impact of TD-AC on Accu and TruthFinder by pairwise comparison of the accuracy values on real datasets Exam with 32 attributes, Stocks and Flights (DCR $\geq$ 66)

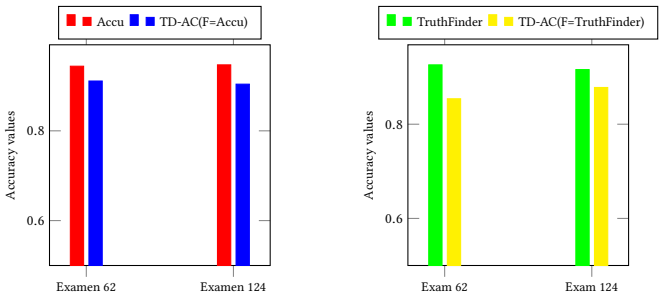


Figure 5: Study of the impact of TD-AC on Accu and TruthFinder by pairwise comparison of the accuracy values on the real datasets Exam with 62 and 124 attributes (DCR $\leq$ 55)

TD-AC outperforms baseline partitioning approaches. TD-AC highly improves the accuracy of **AccuGenPartition** by 1% at least and by 14% at most (see Figure 1) with a time complexity

around 200 less significant (see Tables 4a, 4b and 4c). **AccuGenPartition** is our baseline *brute force* approach proposed in [2] for the truth discovery with attribute partitioning problem with two weighting functions: **Max** and **Avg**. To discover the optimal partition, *k-means* combined with the*silhouette index* has been shown in Table 5 to be better than **Max** and **Avg** because: (i) k-means is a robust partitioning technique with a well-defined optimization strategy; and (ii) silhouette returns the most structurally homogeneous existing clusters. This explains the effectiveness of **TD-AC**. The drastic reduction of the running time with **TD-AC** is because it only requires one iteration to last without exploring all the possible partitions.

TD-AC improves the accuracy of base algorithms. When data attributes are structurally correlated, **TD-AC** significantly enhances the accuracy (from 5 to 35%) of standard algorithms (see Tables 4a, 4b, 4c, 9 and Figure 3). Standard algorithms alone do not capture the structural correlations between attributes leading to biased results. In the cases where the conditions do not match our working setting, **TD-AC** does not degrade the performances of the standard algorithms (see Tables 6 and 7). The impact of **TD-AC** is more important for **Accu** than **TruthFinder** because the former captures better the different levels of reliability of the sources. Such a impact introduces, however, a surplus in terms of execution time which is fortunately reasonable.

Correlation between coverage and TD-AC accuracy. The main observation is that **TD-AC** is more efficient when the data coverage is very high, i.e. DCR $\geq$ 66% (see Figure 4) because more one has in terms of information the better is the clustering with k-means. Lot of missing values, i.e. very sparse truth vectors affect both the quality of the clustering and the truth discovery process (see Figure 5).

5 RELATED WORK

A significant effort has been made in truth discovery area over the past years which has led to several approaches [12, 15]. The simplest approach is the majority vote which considers the truth said by the majority of sources. More elaborated approaches try to model the different levels of reliability of the sources and domain-specific aspects of the truth. For instance, **TruthFinder**[14], one of the first proposed standard algorithms, is a probabilistic model based on Bayesian analysis with similar values supporting each others in vote counts. Methods such as **DEPEN**, **Accu** and **AccuSim**[4] take into consideration copy relationships that may exist between sources by penalizing the vote of a source if it is detected as a copy of another source. **DART** (Domain-AwaRe Truth Discovery) [10] is both a probabilistic and a bayesian model which integrates the domain expertise level. Very recent methods such as [6, 15] capture the correlations between objects in the domain of **Mobile Crowd Sensing**.

The research works that are connexe to our studied problem are [2] and [13]. The proposal in [2] is a brute force approach that explores all the possible partitions of a given set of attributes in order to discover the one maximizing the precision of a standard truth discovery algorithm. The goodness of a partition in this case is based on a weighting function over sources' reliability levels. The work in [13] focuses on object partitioning based on domain knowledge and some additional constraints.

6 CONCLUSION AND PERSPECTIVES

In this work, we have studied the truth discovery problem in a setting where the attributes of the data are structurally correlated. As a solution, we have proposed a new approach, called **TD-AC**, built on an abstract representation of the truth in the data, the k-means clustering technique and the silhouette measure to automatically find an optimal partitioning of the input data (or a near-optimal) maximizing the accuracy of any base truth discovery process. Through an intensive experimental evaluation over various types of datasets, we have then shown the effectiveness and efficiency of **TD-AC** compared to existing partitioning strategies and its positive impact to the accuracy of any standard truth discovery process.

Despites of that, we have noticed that when the dataset contains lot of missing values, the impact of our approach is less significant. This can be explained by the use of sparse truth vectors in the clustering step, making the finding of the optimal partition hard. Moreover, even if the running time of our approach and standard algorithms is reasonable in the presence of small size datasets, it become important when the number of attributes, objects and sources is very large. As research perspectives, we plan to (i) improve our approach to better account for data with lot of missing values on the one hand; and (ii) on the other hand, to propose an optimization of the running time of our approach, in particular the optimal partition computation, by using parallel computation. We also plan to compare ourselves to a larger set of standard truth discovery algorithms and the partitioning approach in [13].

REFERENCES

[1] Mouhamadou Lamine Ba, Laure Berti-Équille, Kushal Shah, and Hossam M. Hammady. 2016. VERA: A Platform for Veracity Estimation over Web Data. In *Proc. International Conference on World Wide Web*. ACM, Montreal, Canada, 159–162.

[2] Mouhamadou Lamine Ba, Roxana Horincar, Pierre Senellart, and Huayu Wu. 2015. Truth Finding with Attribute Partitioning. In *Proc. International Workshop on Web and Databases*. Association for Computing Machinery, New York, USA, 27–33.

[3] Mouhamadou Lamine Ba, Sébastien Montenez, Talel Abdessalem, and Pierre Senellart. 2014. Monitoring moving objects using uncertain web data. In *Proc. International Conference on Advances in Geographic Information Systems*. ACM, Dallas, USA, 565–568.

[4] Xin Dong, Laure Berti-Equille, and Divesh Srivastava. 2009. Integrating Conflicting Data: The Role of Source Dependence. *PVLDB* 2 (08 2009), 550–561.

[5] Xin Luna Dong, Laure Berti-Equille, Yifan Hu, and Divesh Srivastava. 2010. Global detection of complex copying relationships between sources. *Proc. VLDB Endowment* 3, 1-2 (2010), 1358–1369.

[6] Yang Du, Yu-E Sun, He Huang, Liusheng Huang, Hongli Xu, Yu Bao, and Hansong Guo. 2019. Bayesian co-clustering truth discovery for mobile crowd sensing systems. *IEEE Transactions on Industrial Informatics* 16, 2 (2019), 1045–1057.

[7] Alban Galland, Serge Abiteboul, Amélie Marian, and Pierre Senellart. 2010. Corroborating Information from Disagreeing Views. In *Proc. ACM International Conference on Web Search and Data Mining*. Association for Computing Machinery, New York, USA, 131–140.

[8] Eric Gaussier. Accessed Mai 2020.. Partitionement de documents. Clustering, http://ama.imag.fr/~gaussier/Courses/ATD/Clustering.pdf.

[9] Xian Li, Xin Luna Dong, Kenneth Lyons, Weiyi Meng, and Divesh Srivastava. 2012. Truth Finding on the Deep Web: Is the Problem Solved? *Proc. VLDB Endow.* 6, 2 (Dec. 2012), 97–108.

[10] Xueling Lin and Lei Chen. 2018. Domain-aware multi-truth discovery from conflicting sources. *Proc. VLDB Endowment* 11, 5 (2018), 635–647.

[11] Giovanna Menardi. 2011. Density-based Silhouette diagnostics for clustering methods. *Statistics and Computing* 21, 3 (2011), 295–308.

[12] Dalia Attia Waguih and Laure Berti-Equille. 2014. Truth Discovery Algorithms: An Experimental Evaluation. arXiv:cs.DB/1409.6428

[13] Yi Yang, Quan Bai, and Qing Liu. 2019. A probabilistic model for truth discovery with object correlations. *Knowledge-Based Systems* 165 (2019), 360–373.

[14] Xiaoxin Yin, Jiawei Han, and Philip Yu. 2008. Truth Discovery with Multiple Conflicting Information Providers on the Web. *Knowledge and Data Engineering, IEEE Transactions on* 20 (Juil 2008), 796 – 808.

[15] Ming Zhao and Jia Jiao. 2020. Police: An Effective Truth Discovery Method in Intelligent Crowd Sensing. In *Proc. Artificial Intelligence and Security*, Vol. 12239. Springer, Hohhot, China, 384–398.

Short Paper

Towards Automated Concept-based Decision Tree Explanations for CNNs

Radwa Elshawi
University of Tartu
Tartu, Estonia
radwa.elshawi@ut.ee

Youssef Sherif
University of Tartu
Tartu, Estonia
Youssef.sherif@ut.ee

Sherif Sakr
University of Tartu
Tartu, Estonia
sherif.sakr@ut.ee

ABSTRACT

Currently, deep learning models have been widely used in different application domains due to their notable performance. Explaining the decisions made by deep learning models is important for end-users to enable them to comprehend and diagnose the trustworthiness of the model. Most of the current interpretability techniques provide explanations in the form of importance score for the input pixels or features. However, summarizing such importance scores for input features to provide human-interpretable explanations is challenging. To this end, we propose *Automated Concept-based Decision Tree Explanations* (ACDTE), a novel local explanation framework that provides human-understandable and concept-based explanations for classification networks. Our framework provides end users with the flexibility of customizing the explanations by allowing users to provide the dataset in which visual human-understandable concepts are automatically extracted. Then, such concepts are interpreted through a shallow decision tree that includes concepts that are deemed important to the model in predicting the decision of specific instance. In addition, ACDTE generates counterfactual explanations, suggesting the the minimum changes in the instance's concept-based explanation that lead to a different prediction. Our experiments demonstrate that such a shallow decision tree is faithful to the original neural network at low tree depth. The human interpretability of the explanations provided from our framework is evaluated through humans experiments, showing that our framework generates faithful and interpretable explanations.

1 INTRODUCTION

Since deep learning (DL) models have been achieving remarkable success over the last years in different application domains [1, 3], gaining insights into such models' predictions has received great attention over the last few years and in some cases, there is also a legal requirement to do so [7]. Among the various DL models, convolution neural networks (CNN) achieve remarkable performance in different computer vision tasks including self-driving cars and medical diagnoses. A main drawback for DL models, that prevents their wide adoption in critical domains, is their inscrutable nature of their prediction process that makes them black-boxes. Explaining the behaviour of DL models enables humans to understand the model behaviour, and hence, can increase their trust in the model if the decisions made by the model appear reasonable to humans.

There is no agreement among researchers about what would constitute a satisfactory explanation [13]. However, recent studies over 250 papers have concluded that explanations are counterfactual [12, 13]. Techniques for explaining DL models can be broadly partitioned into two main approaches. The first approach is to identify the evidence that the network uses to make a specific prediction by creating a heatmap that identifies the main parts of the image, which are salient to the prediction [16, 19, 21]. The second approach focuses on providing explanations in the form of human-understandable concepts [5, 6, 20]. Instead of assigning an importance score for each pixel or input feature, the explanation comes in the form of important human-understandable concepts that contribute toward the prediction. Understanding how concepts affect a particular model prediction may reveal potential unwanted bias learned by the model.

In this paper, we describe a framework called Automated Concept-based Decision Tree Explanations (ACDTE) to automatically identify high-level human-understandable concepts which are important for the machine learning model for predicting the decision of a specific instance by aggregating related local image segments (concepts) across diverse data and then decompose the evidence for a prediction for image classification into such concepts through an interpretable shallow decision tree. The explanation provided by the ACDTE framework is expressive and provides not only succinct evidence why a particular image has been assigned to a particular class, but also counterfactuals suggesting what is the least number of concepts needed to be changed are, in an instance's explanation, to change the predicted outcome. We summarize our contributions as follows: 1) A novel local explanation framework to provide automatically extracted concept-based explanations for CNNs in the form of important concepts for the prediction of specific instance presented as a shallow interpretable decision tree that is faithful to the black-box model, 2) A counterfactual explanation, suggesting the changes in the important concepts for the prediction of a specific image that lead to a different outcome, 3) Evaluation of the faithfulness of the explanations provided by ACDTE to the black-box model and the quality of the provided explanations. For ensuring repeatability as one of the main targets of this work, we provide access to the source codes and the detailed results for the experiments of our study[1].

The remainder of this paper is organized as follows. In Section 2, we present our proposed technique, ACDTE. We present a detailed experimental evaluation for our proposed techniques in Section 3 before we finally conclude the paper in Section 4

2 METHODS

In the following, we present ACDTE which is a local explanation technique that explains the prediction of a particular image. ACDTE takes a trained classifier, an image to be explained, and a set of images from user-specified dataset as input. It then extracts concepts presented in these images and interpret these concepts through a shallow decision tree that identifies the main concepts

[1] https://github.com/DataSystemsGroupUT/ACDTE

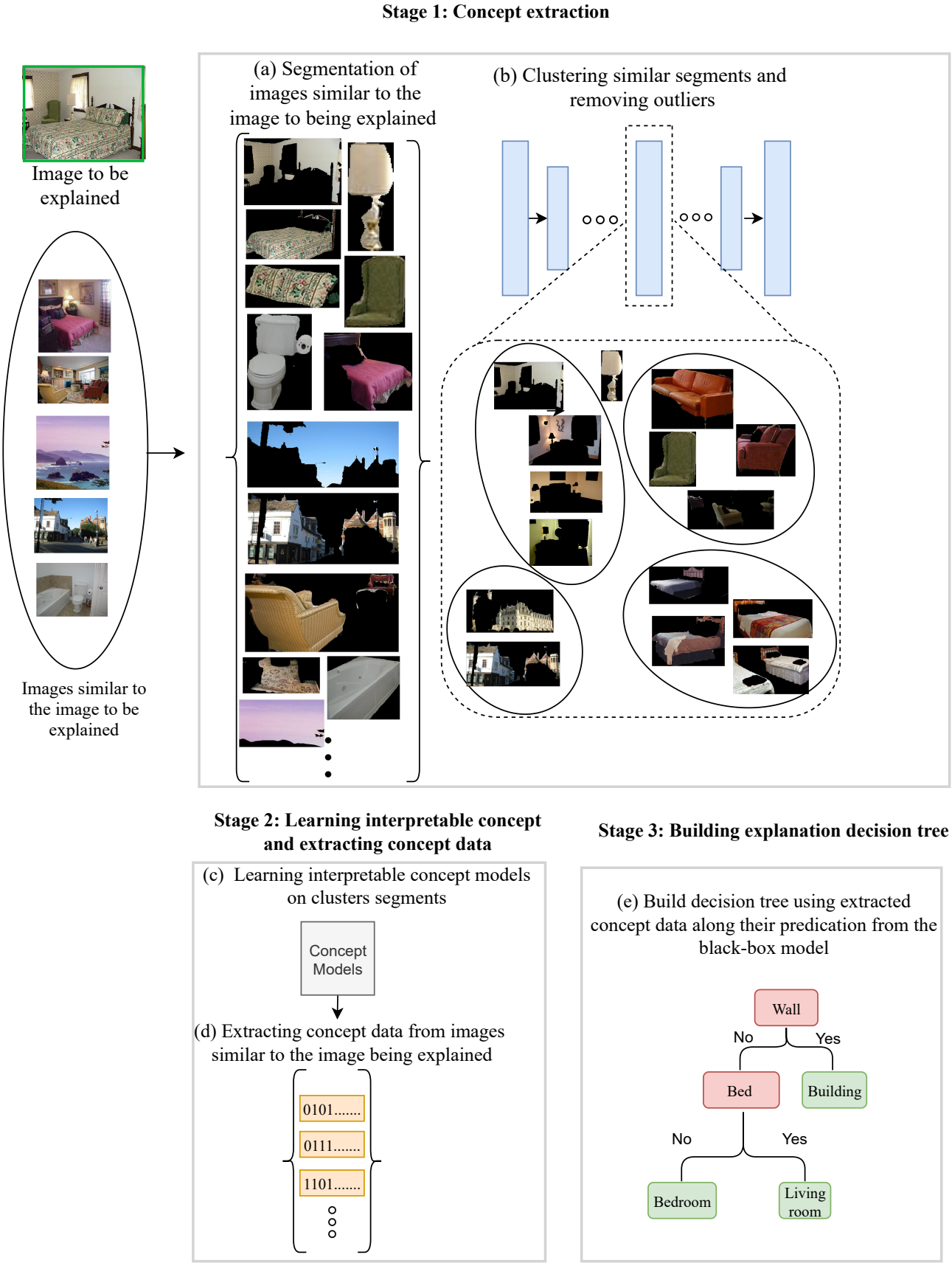


Figure 1: ACDTE pipeline (a) Extract a set of similar images to the image to be explained either from the main task dataset or related dataset. Each image in the selected images is segmented. (b) Segments are clustered in the activation space and outliers are removed to form coherent clusters that represent concepts. (c) Training a linear model for each concept to act as a concept detector. (d) For each image in the activation space, use concepts detectors to form a binary feature vector. (e) Feature vectors along with the prediction of the target network are used to train a shallow decision tree. The decision tree provides a natural explanation for the contributing concepts for the prediction, in addition to counterfactual explanation.

that have deemed important for the prediction of the image being explained, in addition to the minimum number of concepts that need to be changed to alter the prediction of the image to be explained. Figure 1 summarizes the general pipeline for the proposed framework consists of three main phases concept extraction, learning interpretable concept models and extracting concept data, and building explanation decision tree in which explanation based on both decision and counterfactuals, is extracted.

2.1 Phase 1: Concept Extraction

Given a pretrained image classification model m, and the image to be explained I, our framework provides end users with the flexibility of extracting concepts either from the dataset used in the classification task or from a related dataset to the main task dataset. To extract concepts, we choose the top k images similar to I, denoted S. Similarity between I and the set of provided images is defined to be the Euclidian distance between their corresponding activation maps obtained from an intermediate layer from m. In this paper, we use a constant value for $K = 100$, leaving the exploration of different values to future work. To extract concept data, each of the images in S is segmented using semantic image segmentation technique, see Figure 1(a). In order to automate the process of concept extraction, a significant number of studies in literature focused on semantic segmentation algorithms that aim to assign a meaningful class to each pixel [9, 11, 14, 18]. ACDTE uses DeepLabv3+ [2] segmentation technique which has been widely used due to its superior performance on dense datasets (after examining several segmentation techniques). To ensure meaningfulness of the extracted concepts, we cluster segments into a number of clusters such that segments of the same cluster represent a particular concept. In order to automate the process of clustering segments, we define the similarity between segments to be the euclidian distance between their corresponding activation maps obtained from the intermediate layer of model m. Each segment was resized to the original size of m. All segments were then passed through m to obtain their layer presentations. All segments are then clustered using K-means clustering algorithm [10], see Figure 1(b). To ensure meaningfulness of the extracted concepts, we exclude the following two types of clusters: 1) Clusters that have segments that only coming from a single image or a very few number of images. 2) Clusters with segments less than N segments. In this work, we use a constant value for N equals $0.4\sqrt{n_c}$, where n_c is the number of segments in cluster c, leaving the exploration of different values for N to future work. The main problem with clusters of few segments is that the concepts they present are uncommon in the neighborhood of the image being explained. For example, bed segments are present in almost every bedroom image and therefore, are expected to form a coherent cluster while lamp segments are presented in very few bedroom images and hence lamp cluster should be removed. The output of this phase is the final set of clusters representing the learnt concepts denoted $C = \{c_1, ...c_n\}$, where n is the number of clusters after the exclusion criteria.

2.2 Phase 2: Learning Interpretable Concept Models and Extracting Concept Data

For each segment $x \in c$, the hidden layer activations $a = m_l(x)$ at layer l are extracted and stored along its corresponding concept label. For each candidate concept $c \in C$, we train a logistic binary classifier h_c to detect the presence of concept c, see Figure 1(c). Training each concept c is done on dataset D_c, which is mix of segments balancing the presence and absence of concept c. We define $D_c = D_c^+ \cup D_c^-$, where $D_c^+ = \{(m_l(x^1), y_c^1), ..., (m_l(x^{|c|}), y_c^{|c|})|_{y_c=1}\}$ and $D_c^- = \{(m_l(x^1), y_c^1), ..., (m_l(x^{|c|}), y_c^{|c|})|_{y_c=0}\}$, where $y \in \{0, 1\}$ indicates the absence or the presence of concept c in a segment. Negative examples D_c^- for each concept c are selected randomly from other cluster concepts such that the number of examples in D_c^+ and D_c^- are equal. We use these concept classifiers for each image $s \in S$ to create a binary vector $v = (r_1, r_2, ..., r_n)$ representing the presence or absence of each concept $c \in C$ in s, where $r_i = h_{c_i}(s)$, $r_i \in \{0, 1\}$. For each image $s \in S$, we store its class prediction from model m along with its binary concept vector v for training a decision tree, see Figure 1(d).

2.3 Phase 3: Building Concept Decision Tree

Concept vector v predicted for each image $s \in S$ along with the corresponding prediction $m(s)$ are used to train a decision tree T which is intended to mimic the behavior of m locally in the S neighborhood, see Figure 1(e). We use the default implementation of decision tree from *scikit-learn* [15]. The ACDTE approach considers decision tree classifier due to its interpretable nature that allows concept rules to be derived from a root-leaf path in the decision tree, in addition to counterfactuals that can be extracted by symbolic reasoning over a decision tree. Increasing the depth of a decision tree increases the prediction accuracy which leads to less interpretable results as the number of nodes increases exponentially with depth. Thus, a shallow decision is favourable as it is more comprehensible by humans. In this work, we use a fixed depth leaving the exploration of dynamic depth to future work. In order to guarantee fast and easy search for counterfactuals, we consider all possible paths in the decision tree leading to a decision that is not equal to the decision of I. Among all these paths, we only consider the one with the minimum number of spilt conditions that are not satisfied by instance I. As an example, consider the decision tree in Figure 2 explaining the prediction from ResNet50 pretrained on places dataset [22] of an image as a coast. The concepts used in building the decision tree is based on selecting the top 100 images from a random selection of 1000 images from the ADE20k dataset [23]. The left branch of the tree indicates the presence of a concept while the right branch indicates the absence of that concept. The tree gives insights into the main human-understandable concepts from ADE20K dataset that appear important for ResNet50 in predicting the coast image. The decision tree provides a natural explanation for each path. It is clear from the explanation tree that the image has been predicted as a coast because of the existence of the concepts 'mountain' and 'sea'. As a further output, ACDTE computes a counterfactual; we have two counterfactual paths in the decision tree shown in Figure 2. The first one is the presence of 'mountain', absence of 'sea', presence of 'tree' that leads to the prediction of class 'snowy mountain', while the second is the presence of 'mountain', absence of 'sea', absence of 'tree' that leads to the prediction of class 'highway', as shown in Figure 2. Figure 3 shows sample segments of concepts along the explanation path of the coast image shown in Figure 2.

3 EXPERIMENTS AND RESULTS

In this section, we evaluate the meaningfulness of the explanations provided by our framework. In addition, we evaluate the faithfulness of the proposed framework to the black-box model.

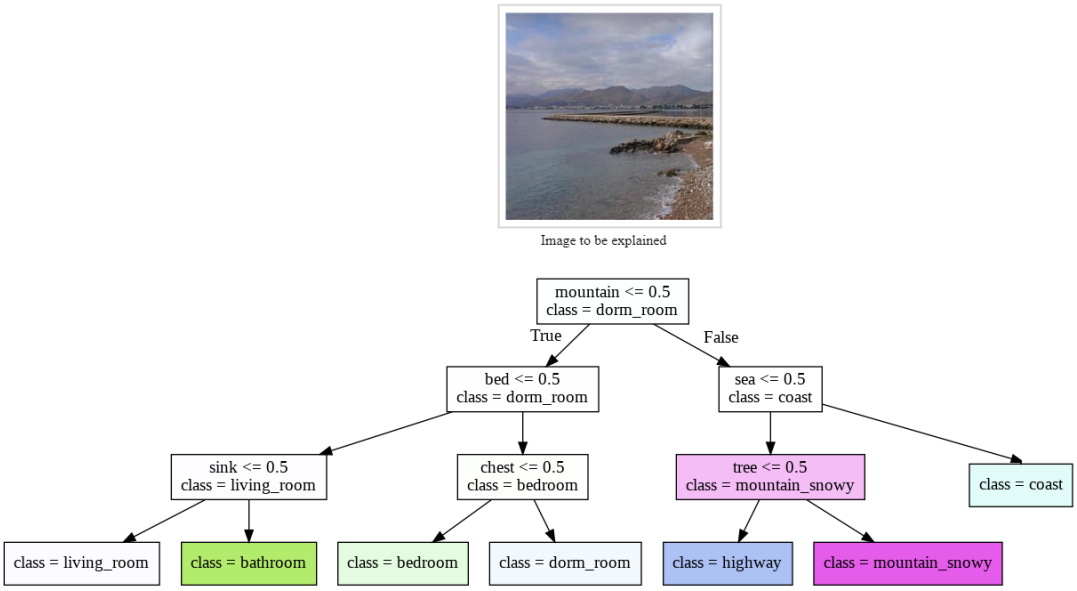


Figure 2: Shallow concept-based explanation decision tree of depth 4 explaining the prediction of coast image.

Figure 3: Sample segments of concepts along the explanation path of the coast image shown in Figure 2. Text below each group of images describes its original class of the extracted concepts.

3.1 Experiment Setup

As an experimental example, we use ACDTE to explain the predictions of the widely-used Resnet50 that has been pre-trained on the places dataset. We select a subset of 30 classes out of the 365 classes from places dataset. We experimented extracting the concepts from ADE20K dataset. More specifically, to explain the prediction of an instance from places dataset, we randomly select 1000 images from ADE20K dataset and extract concepts from the nearest 100 images. To evaluate the performance of the ACDTE, we randomly select 1000 images, denoted X, from the 30 selected classes of the places dataset.

3.2 Are ACDTE Explanations Faithful to the Black-box Model?

We consider the following metrics in evaluating how well the decision tree inferred by ACDTE and the explanations returned mimic the black-box model.

- **fidelity** $\in [0, 1]$: It compares the prediction of the decision tree T and the black-box model m on the set of images S used to train the decision tree [4].
- **hit** $\in \{0, 1\}$: It compares the prediction of the decision tree c and the black-box model m on the instance to be explained I [8]. It returns 1 if $m(I) = T(I)$, and 0 otherwise.

We measure the fidelity by F1-measure [17] and report the aggregated values of the F1 measure across all instances in X at tree depth of 5, 10, 15 and 20, see Figure 4(a). We report hit by averaging its values across the instances in X at tree depth of 5, 10, 15 and 20, see Figure 4(b). The results show that fidelity and hit increases as the tree depth increases, however tree depth of 10 is able to achieve reasonable fidelity of 0.87 and hit of 0.91.

3.3 Examining the Significance of the Extracted Concepts from ACDTE

To confirm the importance of the formed concepts of ACDTE, we run ACDTE on each of the images in X and return the set of clusters obtained from the concept extraction phase. We rank the returned clusters for each image in X according to their compactness that is captured by calculating the average distance between cluster center and each point in the same cluster. The smaller the average distance indicates that the cluster is tightly formed and shows a motion coherent view. The intuition behind that ranking is that compact clusters most likely represent a concept that is frequently present in the neighbourhood of the image to be explained, and hence, have a significant role in forming the decision boundary between classes. For each image in X, we build different decision trees based on excluding the top k concepts obtained from the concept extraction phase, where k=0, 2, 5, 8 and 10. Figure 5 shows the prediction accuracies on the set of images used to train the decision tree when removing the most important k concepts aggregated across all the instances in X. The results show that accuracy decreases significantly from 92.4% to 75.3% when removing the 10 most important concepts which reflects the variable significance of the automatically extracted concepts.

3.4 Concept Classifier Prediction Performance

Concept models performance vary across the different layers of the main task model (ResNet50). In order to identify the best

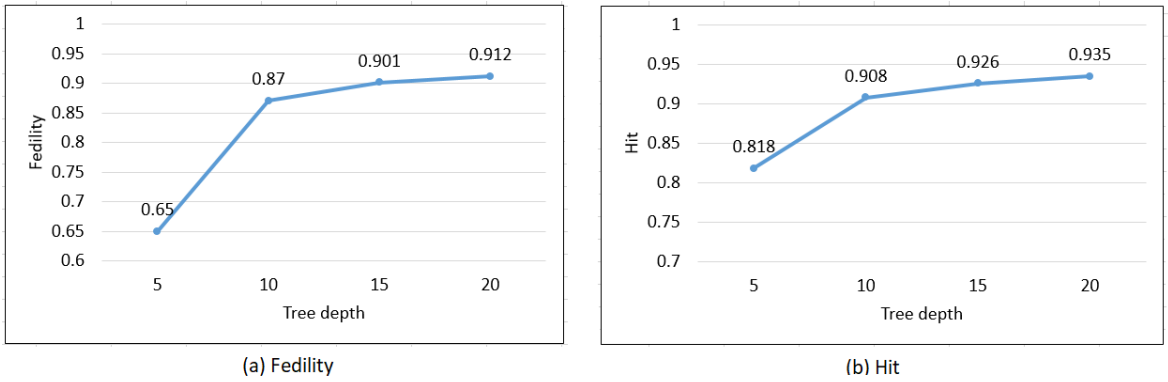


Figure 4: Fidelity and hit at different tree depth

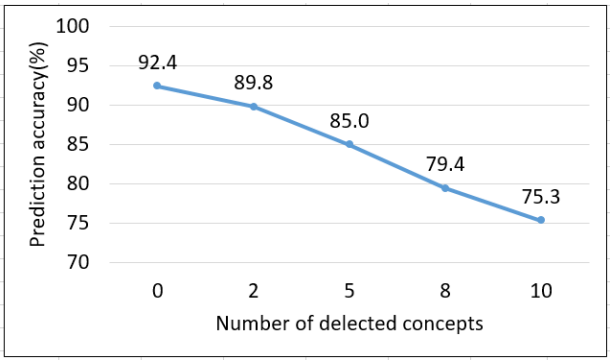


Figure 5: Prediction accuracy of decision tree as removing the top k important concepts aggregated across all the instances in X.

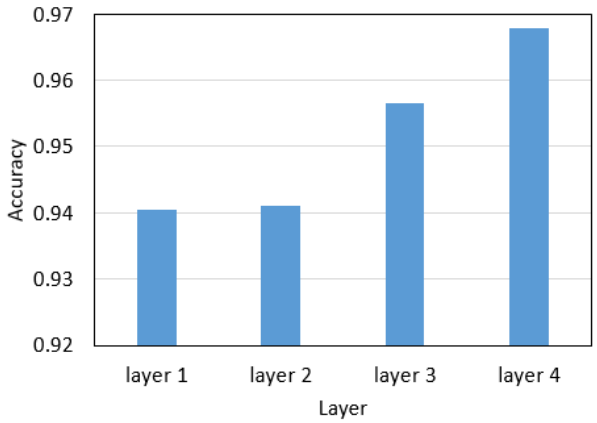


Figure 6: Average accuracy of all concept classifiers trained for the main layers of ResNet50

layer to extract feature vectors used to train concept classifiers, we compare the average accuracy of the concept models built on vectors extracted from the major layers of Resnet50 and report this average accuracy averaged over all instances in X. Major layers refer to the conv2 x (layer1), conv3 x (layer 2), conv4 x (layer 3), and conv5 x (layer 4) blocksections of sublayers of Resnet50. Figure 6 shows that all layers have high average accuracy and the deeper the extraction layer, the higher the accuracy. Figure 6 shows that the average classifiers accuracy was the highest at the fourth layer, achieving an accuracy of 0.97.

3.5 Decision Tree Performance

Figure 7 shows how the accuracy of the decision tree obtained from ACDTE responds to the changes in the maximum tree depth and the layer from which deep features are extracted, to train the concept models. We incrementally increase the depth of the decision tree obtained from ACDTE for each instance in X and change the layer in which features from ResNet are extracted to train the concept models. Then, we measure the prediction accuracy of the instances used to train the decision tree and report this accuracy averaged over all instances in X. Result show that the accuracy improves significantly as more concepts are added and then slightly flatten out as depth increases beyond 15. The results also demonstrates that layer 4 of ResNet50 achieves the best performance in terms of the decision tree prediction accuracy.

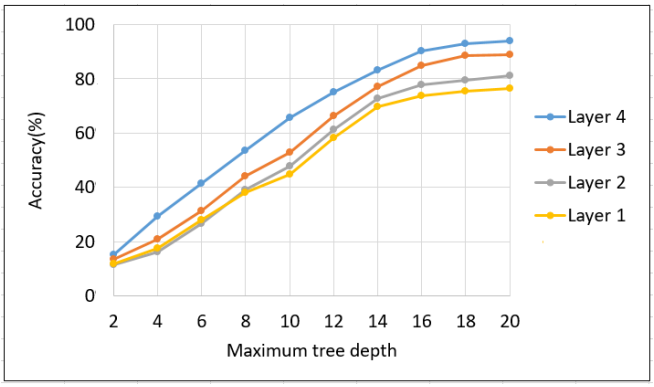


Figure 7: Decision tree accuracy vs. decision tree depth

3.6 Human Evaluation of the Visual Explanations

To measure the meaningfulness of the extracted concepts, we randomly select 50 instances from X and get the concepts used in their explanations. We ask 30 human participants to identify that segments belong to a concept versus a random set of segments. The evaluation interface is shown in Figure 8. Results show that 87% of participants choose the concept segments. To measure the significance of the important concepts extracted from the ACDTE,

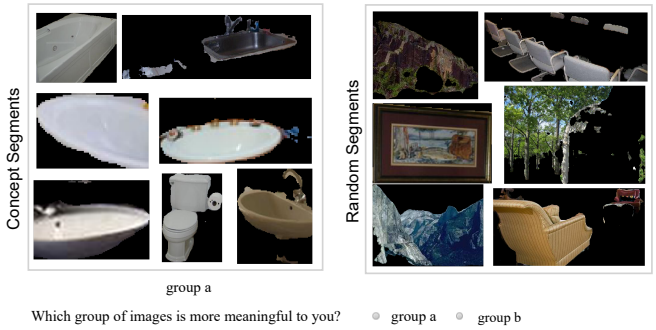


Figure 8: Human evaluation interface for identifying meaningful concepts

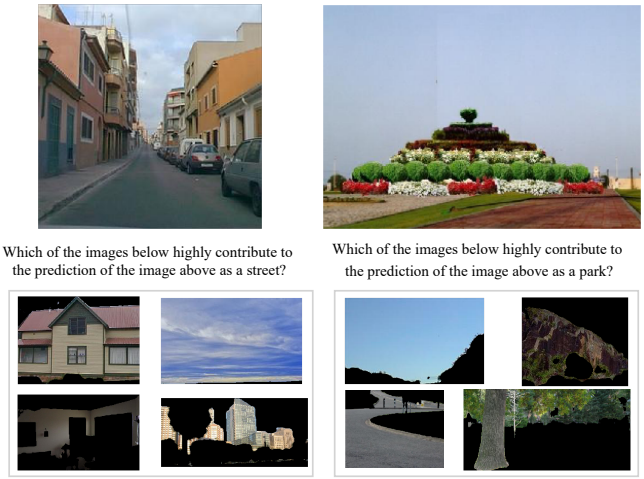


Figure 9: Sample examples of human experiments for choosing the most contributing concept to their predictions

we ask the 30 participants to select the most meaningful concept that contributes to a particular prediction made by ResNet50 for 30 different images. In each task, participants are shown the image to be explained along with its prediction and four concepts in which one of them represents the top concept identified by ACDET for explaining this image and the other three concepts are randomly chosen. Participants are asked to select the most meaningful concept that contribute to the prediction. Figure 9 shows two sample images along with four different concepts in which participants are asked to choose the most contributing concept for the prediction of these images. On average, 85% of the participants chose the concept obtained by ACDTE as the most important concept.

4 CONCLUSION

We introduced ACDTE, a post-training local explanation technique that automatically extract groups of input features from images similar to the images to be explained and group these features into high-level human-understandable concepts. We verified the meaningfulness and coherence of these concepts through human experiments and further validated that these concepts carry some signals indicating to the correct prediction class for the instance to be explained. Representing these concepts in a shallow decision tree allows users to infer which concepts are significant in the prediction of the image to be explained. A future direction of automated concept-based explanation is to consider other types of data such as texts.

ACKNOWLEDGMENT

The work of Sherif Sakr and Youssef Sherif is funded by the European Regional Development Funds via the Mobilitas Plus programme (grant MOBTT75). The work of Radwa Elshawi is funded by the European Regional Development Funds via the Mobilitas Plus programme (MOBJD341).

REFERENCES

[1] Javed Ashraf, Asim D Bakhshi, Nour Moustafa, Hasnat Khurshid, Abdullah Javed, and Amin Beheshti. 2020. Novel Deep Learning-Enabled LSTM Autoencoder Architecture for Discovering Anomalous Events From Intelligent Transportation Systems. *IEEE Transactions on Intelligent Transportation Systems* (2020).

[2] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. 2018. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)*. 801–818.

[3] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. 2015. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

[4] Finale Doshi-Velez and Been Kim. 2017. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608* (2017).

[5] Amirata Ghorbani, James Wexler, James Y Zou, and Been Kim. 2019. Towards automatic concept-based explanations. In *Advances in Neural Information Processing Systems*. 9273–9282.

[6] Abel Gonzalez-Garcia, Davide Modolo, and Vittorio Ferrari. 2018. Do semantic parts emerge in convolutional neural networks? *International Journal of Computer Vision* 126, 5 (2018), 476–494.

[7] Bryce Goodman and Seth Flaxman. 2017. European Union regulations on algorithmic decision-making and a "right to explanation". *AI Magazine* 38, 3 (2017), 50–57.

[8] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Dino Pedreschi, Franco Turini, and Fosca Giannotti. 2018. Local rule-based explanations of black box decision systems. *arXiv preprint arXiv:1805.10820* (2018).

[9] Wei Liu, Andrew Rabinovich, and Alexander C Berg. 2015. Parsenet: Looking wider to see better. *arXiv preprint arXiv:1506.04579* (2015).

[10] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[11] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3431–3440.

[12] Tim Miller. 2018. Contrastive explanation: A structural-model approach. *arXiv preprint arXiv:1811.03163* (2018).

[13] Tim Miller. 2019. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence* 267 (2019), 1–38.

[14] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. 2015. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE international conference on computer vision*. 1520–1528.

[15] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[16] Ramprasaath R Selvaraju, Abhishek Das, Ramakrishna Vedantam, Michael Cogswell, Devi Parikh, and Dhruv Batra. 2016. Grad-CAM: Why did you say that? *arXiv preprint arXiv:1611.07450* (2016).

[17] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2016. *Introduction to data mining*. Pearson Education India.

[18] Xing Wei, Qingxiong Yang, Yihong Gong, Narendra Ahuja, and Ming-Hsuan Yang. 2018. Superpixel hierarchy. *IEEE Transactions on Image Processing* 27, 10 (2018), 4838–4849.

[19] Wencan Zhang, Mariella Dimiccoli, and Brian Y Lim. 2020. Debiased-CAM for bias-agnostic faithful visual explanations of deep convolutional networks. *arXiv preprint arXiv:2012.05567* (2020).

[20] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. 2014. Object detectors emerge in deep scene cnns. *arXiv preprint arXiv:1412.6856* (2014).

[21] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. 2016. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2921–2929.

[22] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Antonio Torralba, and Aude Oliva. 2016. Places: An image database for deep scene understanding. *arXiv preprint arXiv:1610.02055* (2016).

[23] Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. 2017. Scene parsing through ade20k dataset. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 633–641.

Short Paper

Efficient Maintenance of Distance Labelling for Incremental Updates in Large Dynamic Graphs

Muhammad Farhan
Australian National University
Canberra, Australia
muhammad.farhan@anu.edu.au

Qing Wang
Australian National University
Canberra, Australia
qing.wang@anu.edu.au

ABSTRACT

Finding the shortest path distance between an arbitrary pair of vertices is a fundamental problem in graph theory. A tremendous amount of research has been successfully attempted on this problem, most of which is limited to static graphs. Due to the dynamic nature of real-world networks, there is a pressing need to address this problem for dynamic networks undergoing changes. In this paper, we propose an *online incremental* method to efficiently answer distance queries over very large dynamic graphs. Our proposed method incorporates incremental update operations, i.e. edge and vertex additions, into a highly scalable framework of answering distance queries. We theoretically prove the correctness of our method and the preservation of labelling minimality. We have also conducted extensive experiments on 12 large real-world networks to empirically verify the efficiency, scalability, and robustness of our method.

1 INTRODUCTION

Given a very large graph with billions of vertices and edges, how efficiently can we find the shortest path distance between any two vertices? If such a graph is dynamically changing over time (e.g. inserting edges or vertices), how can we not only efficiently but also accurately find the shortest path distance between any two vertices? These questions are intimately related to distance queries on dynamic graphs. As one of the most fundamental operations on graphs, distance queries have a wide range of real-world applications that operate on increasingly large dynamic graphs, such as context-aware search in web graphs [19], social network analysis in social networks [5, 20], management of resources in computer networks [6], and so on. Many of these applications use distance queries as a building block to realise more complicated tasks, and require distance queries to be answered instantly, e.g. in the order of milliseconds.

Previous studies have primarily focused on distance queries on static graphs [1–3, 10, 11, 13, 22], with little attention being paid to dynamics on graphs. To speed up query response time, a key technique is to precompute a data structure called *distance labelling* that satisfies certain properties such as 2-hop cover [8], and then use this data structure to answer distance queries efficiently. However, when a graph dynamically changes, its distance labelling needs to be changed accordingly; otherwise, distance queries may yield overestimated distances. Although it is possible to recompute a distance labelling from scratch, this leads to inefficiency. As shown in Figure 1, the percentage of affected vertices by a single change often ranges from 10^{-5}% to 10% in various real-world networks, recomputing distance labelling from scratch for each single change not only wastes computing resources, but also may generate inaccurate query results during recomputing process. The question arising is thus how to efficiently and accurately change distance labelling on dynamic graphs in order to support distance queries?

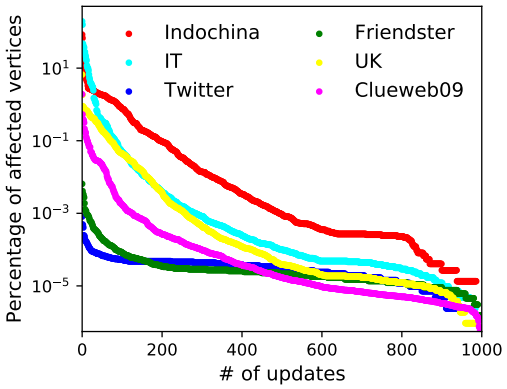


Figure 1: Distribution of affected vertices by a single graph change in various networks, where the results for 1000 graph changes are sorted in the descending order.

In this paper, we aim to develop an *online incremental* method that can dynamically maintain distance labelling on graphs being changed by edge and vertex insertions. Typically, real-world dynamic networks are more vulnerable to insertions than removals and a plethora of such real-world networks are large and frequently updated, primarily accommodating insertions [15, 21]. Thus, an online incremental method for dynamic graphs should possess the following desirable characteristics: (1) *time efficiency* - It can answer distance queries and update distance labelling efficiently (in the order of milliseconds); 2) *space efficiency* - It guarantees the minimum size of distance labelling to reduce storage costs; (3) *scalability* - It can scale to very large networks with billions of vertices and edges.

Challenges. Designing online incremental methods for distance queries on dynamic graphs is known to be challenging [4]. When an edge or a vertex is inserted into a graph, outdated and redundant entries of distance labelling may occur. It was reported that removing such entries is a complicated task [4] because affected vertices need to be precisely identified so as to update their labels without violating the original properties of a distance labelling such as minimality. Further, although query time and update time are both critical for answering distance queries on dynamic graphs, it is not easy (if not impossible) to design a solution that is efficient in both. This requires us to find new insights into dynamic properties of a distance labelling, as well as a good trade-off between query time and update time. Last but not least, scaling distance queries to dynamic graphs with billions of nodes and edges is hard. Previous work [4, 12] mostly considered 2-hop labelling, which has very high space requirements and index construction time; as a result, their query and update performance

are dramatically degraded on large-scale dynamic graphs. Ideally, the labelling size of a graph should be much smaller than its original size. However, the state-of-the-art distance labelling technique, i.e. pruned landmark labeling method (PLL) [4], still yields a distance labelling whose size is 20-30 times larger than the original size of a dataset.

Contributions. Our contributions are summarised as follows:

- Our method overcomes the challenge of eliminating outdated and redundant distance entries. None of the previous studies have addressed this challenge because detecting those entries is too costly [4, 9]. When an edge or a vertex is inserted, previous studies only add new distance entries or modify existing distance entries. This would however lead to an ever increasing size of labelling, particularly when a graph is frequently updated by newly added edges or vertices. Accordingly, both query performance and space efficiency would deteriorate over time.
- We prove the correctness of our proposed method and show that it preserves the desirable property of minimality on our distance labelling. Due to a property called highway cover [10], the minimal size of a distance labelling in this work is much smaller than the size of a 2-hop labelling in previous work [4, 12]. Preserving minimality on a distance labelling thus improves space efficiency and query performance, as well as update performance. We also provide a complexity analysis of our proposed method.
- We conducted experiments using 12 real-world large networks across different domains to show the efficiency, scalability and robustness of our method. Particularly, our method can perform updates under one second, on average, even on billion-scale networks, while still answering queries efficiently in the order of milliseconds and guaranteeing the labelling size of a graph to be much smaller.

2 RELATED WORK

Answering shortest-path distance queries in graphs has been an active research topic for many years. Traditionally, a distance query can be answered using Dijkstra's algorithm [18] on positively weighted graphs or Breadth-First Search (BFS) algorithm on unweighted graphs. However, these traditional algorithms fail to achieve desired response time for distance queries on large graphs. Later, labelling-based methods have emerged as an attractive way of accelerating response time to distance queries [1–3, 8, 10, 11, 13], among which Akiba et al. [3] proposed a pruned landmark labeling (PLL) to precompute a 2-hop cover distance labelling [8]. This method serves as the state-of-the-art for labelling-based distance queries and can handle graphs with hundreds of millions of edges.

So far, only a few attempts have been made to study distance queries over dynamic graphs [4, 12], which are all based on the idea of 2-hop distance labelling or its variants. Akiba et al. [4] studied the problem of updating a pruned landmark labelling for incremental updates (i.e. vertex additions and edge additions). This work however does not remove redundant entries in distance labels because the authors considered that detecting such outdated entries is too costly. This inevitably breaks the minimality of pruned landmark labelling, leading to an ever increase of labelling size and deteriorated query performance over time. To accelerate shortest-path distance queries on large networks, another line of research is to combine a partial distance labelling with online shortest-path searches. Hayashi et al. [12] proposed a fully dynamic approach that selects a small set of landmarks R and precompute a shortest-path tree (SPT) rooted at each $r \in R$. Then, an online search is conducted on a sparsified graph under an upper distance bound being computed via the SPTs. Nevertheless, this method still fails to construct labelling on networks with billions of vertices. Following the same line, a recent work by Farhan et al. [10] introduced a highway-cover labelling method (HL), which can provide fast response time (milliseconds) for distance queries even on billion-scale graphs. However, this approach only works for static graphs.

3 PROBLEM FORMULATION

Let $G = (V, E)$ be an undirected graph where V is a set of vertices and E is a set of edges. We denote by $N(v)$ the set of neighbors of a vertex $v \in V$, i.e. $N(v) = \{u \in V | (u, v) \in E\}$. Given two vertices u and v in G, the *distance* between u and v, denoted as $d_G(u, v)$, is the length of the shortest path from u to v. If there does not exist a path from u to v, then $d_G(u, v) = \infty$. We use $P_G(u, v)$ to denote the set of all shortest paths between u and v in G. Given a graph $G = (V, E)$, an *edge insertion* is to add an edge (a, b) into G where $\{a, b\} \subseteq V$ and $(a, b) \notin E$. Accordingly, a *node insertion* is to add a new node into G together with a set of edge insertions that connect v to existing vertices in G. The following fact is critical for designing algorithms for an edge insertion.

Fact 3.1. *Let $G' = (V, E \cup \{(u, v)\})$ be the graph after inserting an edge (u, v) into $G = (V, E)$. Then for any two vertices $s, t \in V$, $d_G(s, t) \geq d_{G'}(s, t)$.*

That is, the distance between any two vertices never increases after inserting edges or vertices in a graph.

Highway cover labelling. Unlike the previous work [4, 9, 12] that uses 2-hop cover labelling [8], we develop our method using a highly scalable labelling approach, called *highway cover labelling* [10]. Let $R \subseteq V$ be a small set of *landmarks* in a graph $G = (V, E)$. For each vertex $v \in V$, the *label* of v is a set of *distance entries* $L(v) = \{(r_1, \delta_L(r_1, v)), \ldots, (r_n, \delta_L(r_n, v))\}$, where $r_i \in R$ and $\delta_L(r_i, v) = d_G(r_i, v)$. We call $L = \{L(v)\}_{v \in V}$ a *distance labelling* over G whose *size* is defined as: $size(L) = \sum_{v \in V} |L(v)|$. A *highway* $H = (R, \delta_H)$ consists of a set R of landmarks and a distance decoding function $\delta_H : R \times R \rightarrow \mathbb{N}^+$ such that, for any two landmarks $r_1, r_2 \in R$, $\delta_H(r_1, r_2) = d_G(r_1, r_2)$ holds.

Definition 3.2. A *highway cover labelling* is a pair $\Gamma = (H, L)$ where H is a highway and L is a distance labelling s.t. for any vertex $v \in V \backslash R$ and $r \in R$, we have:

$$d_G(r, v) = \min\{\delta_L(r_i, v) + \delta_H(r, r_i) | (r_i, \delta_L(r_i, v)) \in L(v)\}. \quad (1)$$

Highway cover labelling enjoys several nice theoretical properties, such as minimality and order independence. A minimal highway cover labelling can be efficiently constructed, independently of the order of applying landmarks [10].

Given a highway cover labeling $\Gamma = (H, L)$, an upper bound on the distance between any two vertices $u, v \in V \backslash R$ is computed:

$$d^{\top}_{uv} = \min\{\delta_L(r_i, u) + \delta_H(r_i, r_j) + \delta_L(r_j, v) | (r_i, \delta_L(r_i, u)) \in L(u), (r_j, \delta_L(r_j, v)) \in L(v)\} \quad (2)$$

An exact distance query $Q(u, v, \Gamma)$ can be answered by conducting a distance-bounded shortest-path search over a sparsified graph $G[V \backslash R]$ (i.e., removing all landmarks in R from G) under the upper bound $d^{\top}_{uv}$ such that:

$$Q(u, v, \Gamma) = \begin{cases} d_{G[V \backslash R]}(u, v) & \text{if } d_{G[V \backslash R]}(u, v) \leq d^{\top}_{uv}, \\ d^{\top}_{uv} & \text{otherwise.} \end{cases}$$

Problem definition. In this work, we study the problem of answering distance queries over a graph that is dynamically changed by edge and vertex insertions over time. Since a vertex insertion can be treated as a set of edge insertions, without loss of generality, below we define the problem based on edge insertions.

Definition 3.3. Let $G \hookrightarrow G'$ denote that a graph G is changed to a graph G' by an edge insertion. The *dynamic distance querying* problem is, given any two vertices u and v in the changed graph G', to efficiently compute the distance $d_{G'}(u, v)$.

4 ONLINE INCREMENTAL ALGORITHM

In this section, we propose an algorithm IncHL$^+$ to incrementally update labelling to reflect graph changes. Algorithm 1 describes the main steps of IncHL$^+$. Below, we discuss them in detail.

4.1 Finding Affected Vertices

When an update operation occurs on a graph $G = (V, E)$, there exists a subset of "affected" vertices in V whose labels need to be updated as a consequence of this update operation on the graph.

Definition 4.1. A vertex $v \in V$ is *affected* by $G \hookrightarrow G'$ iff $P_G(v, r) \neq P_{G'}(v, r)$ for at least one $r \in R$; *unaffected* otherwise.

We use Λ_r to denote the set of all affected vertices w.r.t. a landmark r and $\Lambda = \bigcup_{r \in R} \Lambda_r$ the set of all affected vertices.

Example 4.2. Consider Figure 2(a) in which 0 and 10 are two landmarks. After inserting an edge $(2, 5)$, $\Lambda_0 = \{5, 8, 9, 10, 13, 14\}$ in Figure 2(b) and $\Lambda_{10} = \{0, 1, 2\}$ in Figure 2(d).

The following lemma states how affected vertices relate to an edge being inserted.

Lemma 4.3. *When $G \hookrightarrow G'$ for an edge insertion (a, b), a vertex $v \in \Lambda_r$ iff there exists a shortest path between v and r in G' passing through (a, b).*

Following Lemma 4.3, we can reduce the search space of affected vertices by eliminating landmarks r with $d_G(r, a) = d_G(r, b)$ since $\Lambda_r = \emptyset$ in such a case. Thus, we assume that $d_G(r, b) > d_G(r, a)$ w.r.t. a landmark r in the rest of this section w.l.o.g. Further, by the lemma below, we can also reduce the search space by "jumping" from the root of a BFS to vertex b.

Lemma 4.4. *When $G \hookrightarrow G'$ with an inserted edge (a, b), we have $d_G(v, r) \geq d_G(a, r) + 1$ for any affected vertex $v \in \Lambda_r$.*

Proof. By Lemma 4.3, there exists a shortest path from any affected vertex v to r going through the edge (a, b) and thus through a. Since a is unaffected and the distance from a to v is equal to or greater than 1, $d_G(v, r) \geq d_G(a, r) + 1$ thus holds. □

Algorithm 2 describes our algorithm for finding affected vertices. Given a graph G with an inserted edge (a, b) and a highway cover labelling $\Gamma = (H, L)$ over G, we conduct a *jumped* BFS w.r.t. a landmark r starting from the vertex b with its new depth $\pi = Q(r, a, \Gamma) + 1$ (Lines 3-4). For every $(v, \pi) \in Q$, we enqueue all the neighbors of v that are affected into Q with new distances $\pi + 1$ (Lines 7-8) and add v to Λ_r as affected vertex (Line 9). This process continues until Q is empty.

Example 4.5. Figure 2 illustrates how our algorithm finds affected vertices as a result of inserting an edge $(2, 5)$. The BFS rooted at landmark 0 is depicted in Figure 2(b), which jumps to vertex 5 and finds six affected vertices $\{5, 8, 9, 10, 13, 14\}$. Similarly, the BFS rooted at landmark 10 is depicted in Figure 2(d), which jumps to vertex 2 and finds three affected vertices $\{0, 1, 2\}$.

Algorithm 1: Incremental algorithm (IncHL$^+$).

```
Input: G, G', (a, b), Γ = (H, L)
Output: Γ' = (H', L')
1 foreach r ∈ R do
2     Λ_r ← FindAffected(G, (a, b), r, Γ)
3     RepairAffected(G', (a, b), Λ_r, r, Γ)
```

Algorithm 2: Finding affected vertices.

```
1  Function FindAffected(G, (a, b), r, Γ)
2      Q ← ∅, Λ_r ← ∅
3      π ← Q(r, a, Γ) + 1
4      Enqueue (b, π) to Q
5      while Q is not empty do
6          Dequeue (v, π) from Q
7          foreach w ∈ N(v) s.t. Q(r, w, Γ) ≥ π + 1 do
8              Enqueue (w, π + 1) to Q
9          Λ_r = Λ_r ∪ {v}
10     return Λ_r
```

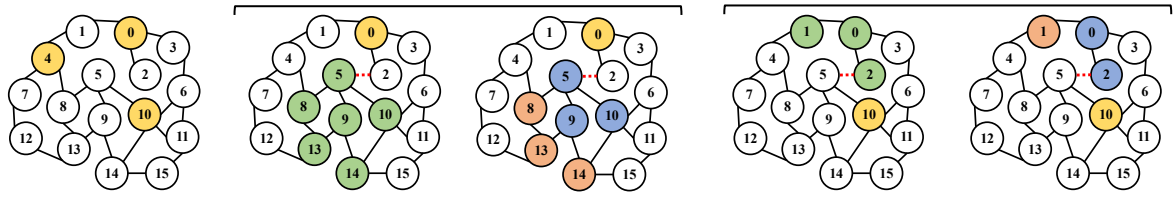
4.2 Repairing Affected Vertices

Now we propose a repair strategy to efficiently update the labels of affected vertices in order to reflect graph changes. The key idea is that, instead of conducting a full BFS on all vertices, we conduct a partial BFS from b only on affected vertices. Further, to avoid unnecessary computations, we distinguish two kinds of affected vertices: (1) affected vertices that are *covered* by other landmarks and can thus be easily repaired by removing an entry from their labels; (2) affected vertices whose labels need to be repaired with accurately calculated distances on a changed graph. The following lemma characterizes the first kind according to the definition of highway cover labelling.

Lemma 4.6. *An affected vertex $v \in \Lambda_r$ is* covered *by a landmark $r' \in R \backslash \{r\}$ iff r' exists in $P_{G'}(v, r)$. If an affected vertex $v \in \Lambda_r$ is covered by r', then any affected vertex $v' \in \Lambda_r$ satisfying $d_{G'}(r, v') = d_{G'}(r, v) + d_{G'}(v, v')$ must also be covered by r'.*

By Lemma 4.6, we can efficiently repair affected vertices $v \in \Lambda_r$ as follows. If v is covered by a landmark $r' \in R \backslash \{r\}$ (i.e., one of the unaffected parents of v does not contain r in its label) and is also a landmark, we only update the highway; otherwise, we remove the entry of r from $L(v)$. If v is not covered by any $r' \in R \backslash \{r\}$, we add/modify the entry of r in $L(v)$. If v is a descendant of covered vertices, we simply remove the entry of r from $L(v)$ (if exists).

Algorithm 3 describes our algorithm for repairing affected vertices. Given a graph G with an inserted edge (a, b) and a set of affected vertices Λ_r, we conduct a BFS w.r.t. a landmark r starting from the vertex b with its new distance $\pi = d_G(r, a) + 1$ (Lines 3-4). We use two queues $Q_{uncovered}$ and $Q_{covered}$ to process uncovered and covered vertices, respectively. If b is covered, we enqueue (b, π) to $Q_{covered}$ and remove the entry of r from the labels of affected vertices (Line 25). Otherwise, we enqueue (b, π) to $Q_{uncovered}$ and start processing vertices in $Q_{uncovered}$ (Line 5). For each vertex $v \in Q_{uncovered}$ at depth π, we examine its affected neighbors w at depth $\pi + 1$. If w is covered, then if w is a landmark, we update the highway (Line 10); otherwise we remove the entry of r from $L(w)$ (Line 12) because there must exist another landmark in the shortest path from w to r and add



(a) (b) (c) (d) (e)

Figure 2: An illustration of our online incremental algorithm IncHL$^+$: (a) a graph with three landmarks 0, 4 and 10 (colored in yellow); (b) and (d) the BFSs for finding affected vertices (colored in green) w.r.t. landmarks 0 and 10, respectively; (c) and (e) the BFSs for repairing affected vertices w.r.t. landmarks 0 and 10, respectively, where vertices with added/modified entries are colored in blue, and vertices with removed entries are colored in red.

Algorithm 3: Repairing affected vertices.

```
1  Function RepairAffected(G', (a, b), Λ_r, r, Γ)
2      Q_uncovered ← ∅, Q_covered ← ∅
3      π ← d_G(r, a) + 1
4      Enqueue (b, π) to Q_covered if covered; otherwise to
         Q_uncovered
5      while Q_uncovered is not empty do
6          while (v, π) ∈ Q_uncovered at depth π do
7              forall w ∈ N(v) s.t. w ∈ Λ_r at depth π + 1 do
8                  if covered(w, π + 1) then
9                      if w is a landmark then
10                         δ_H(r, w) ← π + 1
11                     else
12                         Remove r from L(w)
13                     Enqueue (w, π + 1) to Q_covered
14                 else
15                     Add/Modify {(r, π + 1)} in L(w)
16                     Enqueue (w, π + 1) to Q_uncovered
17                 Remove w from Λ_r
18             Dequeue (v, π) from Q_uncovered
19         while (v, π) ∈ Q_covered at depth π do
20             forall w ∈ N(v) s.t. w ∈ Λ_r at depth π + 1 do
21                 Remove r from L(w)
22                 Remove w from Λ_r
23                 Enqueue (w, π + 1) to Q_covered
24             Dequeue (v, π) from Q_covered
25     Remove entry r from remaining vertices in Q_covered
```

$(w, \pi+1)$ to $Q_{covered}$ (Line 13). Otherwise, we add/modify the entry of r with the new distance $\pi+1$ in $L(w)$ and enqueue w to $Q_{uncovered}$ (Lines 15-16). After that, we remove w from Λ_r (line 17). Then, for each $(v, \pi) \in Q_{covered}$, we remove r from the labels of affected neighbors of v, remove these affected vertices from Λ_r and enqueue them to $Q_{covered}$ (Lines 19-24). We process these two queues, one after the other, until $Q_{uncovered}$ is empty. Finally, we remove the entry of r from the labels of the remaining vertices in $Q_{covered}$ (Line 25).

Example 4.7. Figure 2 illustrates how our algorithm repairs labels as a result of inserting an edge $(2, 5)$. The BFS for landmark 0 is depicted in Figure 2(c), which jumps to vertex 5 and repairs three affected vertices $\{5, 9, 10\}$. The vertices $\{8, 13, 14\}$ are covered by landmarks 4 and 10. Similarly, the BFS for landmark 10 is depicted in Figure 2(e), in which vertices $\{0, 2\}$ are repaired and vertex 1 is covered by landmarks 0 and 4.

5 THEORETICAL RESULTS

Proof of correctness. For $G \hookrightarrow G'$ where our method IncHL$^+$ updates a highway cover labelling Γ over G into a highway cover labelling Γ' over G', we consider IncHL$^+$ to be *correct* iff, whenever $Q(u, v, \Gamma) = d_G(u, v)$ holds for any two vertices u and v in G, then $Q(u', v', \Gamma') = d_{G'}(u', v')$ also holds for any two vertices u' and v' in G'. We prove the theorem below for IncHL$^+$.

Theorem 5.1. *IncHL$^+$ is correct.*

Proof. First, we prove that `FindAffected` returns the set of all affected vertices Λ_r as a result of an edge insertion. IncHL$^+$ (Lines 7-8 of Algorithm 2) guarantees that any vertex being added to Q has one shortest path to a landmark r which goes through the inserted edge (a, b). By Lemma 4.3, such vertices are affected vertices, and thus a vertex v is added to Q in Algorithm 2 iff $v \in \Lambda_r$. Then, we prove that `RepairAffected` repairs $\Gamma = (H, L)$ s.t. (1) $(r, d_{G'}(r, v)) \in L(v)$ for $v \in \Lambda_r$, iff $P_{G'}(r, v)$ contains only one landmark r; (2) $\delta_H(r, r') = d_{G'}(r, r')$ for any $r' \in R \backslash \{r\}$. Starting from b with new distance π, the distances of affected vertices in Λ_r are iteratively inferred on G' and reflected into their labels via $Q_{uncovered}$ if these affected vertices are not covered (Lines 15-16 of Algorithm 3). If an affected vertex v is covered, it is kept in $Q_{covered}$; if v is also a landmark, $\delta_H(r, v)$ in H is updated (Lines 9-10). Thus, the distance entry of r is removed from the labels of affected vertices appearing in $Q_{covered}$, whereas any vertex v appearing in $Q_{uncovered}$ must have $(r, d_{G'}(r, v)) \in L(v)$. □

Preservation of minimality. It has been reported in [10] that, given a graph G, a minimal highway cover labelling $\Gamma = (H, L)$ of G can be constructed using an algorithm proposed in their work, i.e., $size(L') \geq size(L)$ holds for any $\Gamma' = (H, L')$ of G. For $G \hookrightarrow G'$ where IncHL$^+$ updates Γ over G into Γ' over G', we prove that IncHL$^+$ preserves the minimality of labelling.

Theorem 5.2. *If Γ is minimal on G, then Γ' is minimal on G'.*

Proof. By Lemma 4.6, $(r, d_{G'}(r, v)) \in L(v)$ for $v \in \Lambda_r$ iff $P_{G'}(r, v)$ does not contain any other landmark $R \backslash \{r\}$; otherwise we remove the entry of r from the label of v (Line 12, 21 and 25 of Algorithm 3). Thus, the labels of all affected vertices must be minimal after applying IncHL$^+$. For unaffected vertices, their labels should remain unchanged. Hence, Γ' must be minimal. □

Complexity analysis. Let m be the total number of affected vertices, l be the average size of labels (i.e. $l = size(L)/|V|$), and d be the average degree of vertices. For a landmark, Algorithm 2 takes $O(mdl)$ time to find all affected vertices and Algorithm 3 takes $O(md)$ to repair the labels of all affected vertices. We omit l from $O(md)$ for Algorithm 3 because distances for all unaffected neighbors of affected vertices are stored in Algorithm 2. Therefore, IncHL$^+$ has time complexity $O(|R| \times mdl)$. In our

Table 1: Comparing the update time, query time and labelling size of our method with the baseline methods.

Dataset	Update Time (ms)			Query Time (ms)			Labelling Size		
	IncHL$^+$	IncFD	IncPLL	IncHL$^+$	IncFD	IncPLL	IncHL$^+$	IncFD	IncPLL
Skitter	0.194	0.444	2.05	0.027	0.019	0.047	42 MB	153 MB	2.44 GB
Flickr	0.006	0.074	1.73	0.007	0.012	0.064	34 MB	152 MB	3.69 GB
Hollywood	0.031	0.101	48	0.027	0.037	0.109	27 MB	263 MB	12.58 GB
Orkut	2.026	2.049	-	0.101	0.103	-	70 MB	711 MB	-
Enwiki	0.134	0.163	5.91	0.054	0.035	0.071	82 MB	608 MB	12.57 GB
Livejournal	0.245	0.268	-	0.044	0.046	-	122 MB	663 MB	-
Indochina	5.443	158	2018	0.737	0.839	0.063	81 MB	838 MB	18.64 GB
IT	95.92	224	-	1.069	1.013	-	854 MB	4.74 GB	-
Twitter	0.027	0.134	-	0.863	0.177	-	1.14 GB	3.83 GB	-
Friendster	0.159	0.419	-	0.814	0.904	-	2.43 GB	9.14 GB	-
UK	11.49	384	-	3.443	5.858	-	1.78 GB	11.8 GB	-
Clueweb09	40.68	-	-	16.93	-	-	163 GB	-	-

Table 2: Summary of datasets.

Dataset	Network	$\|V\|$	$\|E\|$	avg. deg	avg. dist
Skitter	comp (u)	1.7M	11M	13.081	5.1
Flickr	social (u)	1.7M	16M	18.133	5.3
Hollywood	social (u)	1.1M	114M	98.913	3.9
Orkut	social (u)	3.1M	117M	76.281	4.2
Enwiki	social (d)	4.2M	101M	43.746	3.4
Livejournal	social (d)	4.8M	69M	17.679	5.6
Indochina	web (d)	7.4M	194M	40.725	7.7
IT	web (d)	41M	1.2B	49.768	7.0
Twitter	social (d)	42M	1.5B	57.741	3.6
Friendster	social (u)	66M	1.8B	55.056	5.0
UK	web (d)	106M	3.7B	62.772	6.9
Clueweb09	web (d)	1.7B	7.8B	9.27	7.4

experiments, we notice that m is usually orders of magnitudes smaller than $|V|$ and l is also significantly smaller than $|R|$.

Directed and weighted graphs. For directed graphs, we can store sets of forward and backward labels, namely $L_f(v)$ and $L_b(v)$, for each vertex v which contain pairs $(r_i, \delta_{r_i v})$ from forward and backward BFSs w.r.t. each landmark. Accordingly, we can store forward and backward highways H_f and H_b. Then, we conduct two BFSs to update these labels and highways: one in the forward direction and the other in the backward direction. Our method can also be easily extended to handling weighted graphs by using Dijkstra's algorithm instead of BFSs.

6 EXPERIMENTS

We have evaluated our method to answer the following questions: (Q1) How efficiently can our method perform against state-of-the-art methods? (Q2) How does the number of landmarks affect the performance of our method? (Q3) How does our method scale to perform updates occurring rapidly in large dynamic networks?

Datasets. We used 12 large real-world networks as detailed in Table 2. These networks are accessible at Stanford Network Analysis Project [16], Laboratory for web Algorithmics [7], Koblenz Network Collection [14], and Network Repository [17]. We treated these networks as undirected and unweighted graphs.

Updates and queries. For each network, we randomly sampled 1,000 pairs of vertices as edge insertions, denoted as E_I, where $E_I \cap E = \emptyset$ to evaluate the average update time. Further, we evaluate the average query time with 100,000 randomly sampled pairs of vertices from each network and report the labelling size after reflecting all the updates.

Baseline methods. We compared our method (IncHL$^+$) with the state-of-the-art methods: (1) IncPLL: an online incremental algorithm proposed in [4] which is based on the 2-hop cover labelling to answer distance queries; (2) IncFD: an online incremental algorithm proposed in [12] which combines a 2-hop cover labelling with a graph traversal algorithm to answer distance queries. The codes of these methods were provided by their authors and implemented in C++. We used the same parameter settings for these methods as suggested by their authors unless otherwise stated. For a fair comparison, following [12] we set $|R| = 20$ for IncFD and our methods, except for Clueweb09 which has $|R| = 150$ due to its billion-scale vertices. Our methods were implemented in C++11 and compiled using gcc 5.5.0 with the -O3 option. We performed all the experiments using a single thread on Linux server (Intel Xeon W-2175 with 2.50GHz and 512GB of main memory).

6.1 Performance Comparison

6.1.1 Update Time. Table 1 shows that the average update time of our method IncHL$^+$ outperforms the state-of-the-art methods IncFD and IncPLL on all datasets. This is due to a novel repair strategy utilized by IncHL$^+$. Further, only IncHL$^+$ can scale to very large networks with billions of vertices and edges. IncFD fails to scale to Clueweb09, and IncPLL fails for 7 out of 12 datasets due to very high preprocessing time and memory requirements.

6.1.2 Labelling Size. From Table 1, we see that IncHL$^+$ has significantly smaller labelling sizes than IncFD and IncPLL. When updates occur on a graph, the labelling sizes of IncFD and IncHL$^+$ remain stable because their average label sizes are bounded by the size of landmarks set (i.e. $|R|$). Moreover, IncFD stores complete shortest path trees w.r.t. landmarks; while IncHL$^+$ stores pruned shortest-path trees which lead to labelling of much smaller sizes than IncFD. For IncPLL, the labelling sizes may increase because IncPLL does not remove outdated and redundant entries.

6.1.3 Query Time. In Table 1 the query times of IncHL$^+$ are comparable with IncFD and IncPLL. It has been shown in [9] that query time depends on labelling size. As discussed in Section 6.1.2, the update operations do not considerably affect the labelling sizes of IncFD and IncHL$^+$, and thus their query times remain stable. However, the query times for IncPLL may increase over time because of the presence of outdated and redundant entries, which result in labelling of increasing size.

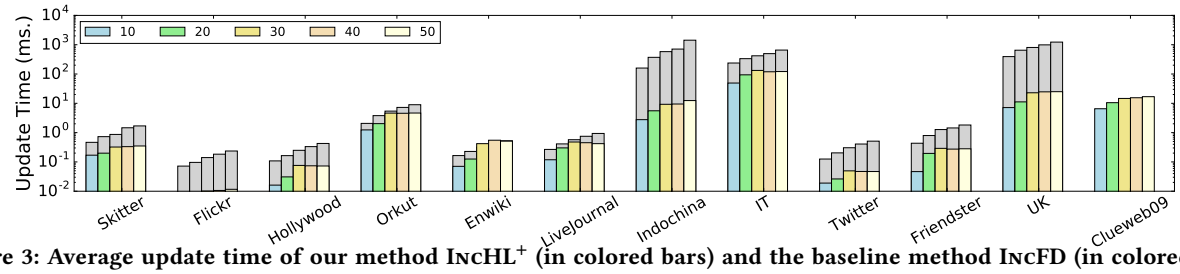


Figure 3: Average update time of our method IncHL$^+$ (in colored bars) and the baseline method IncFD (in colored plus grey bars) under 10-50 landmarks. There are no results of IncFD for Clueweb09 due to the scalability issue.

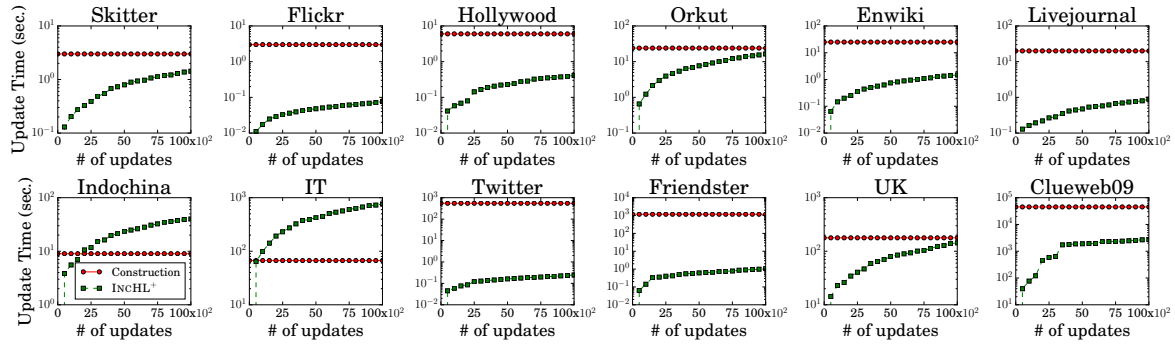


Figure 4: Update time of IncHL$^+$ for performing up to 10,000 updates against construction time of labelling from scratch.

6.2 Performance with Varying Landmarks

Figure 3 shows the average update time of our method IncHL$^+$ against the baseline method IncFD under varying landmarks, i.e., $|R| \in [10, 20, 30, 40, 50]$. As we can see, IncHL$^+$ outperforms IncFD on all the datasets against almost every selection of landmarks. We can also see the performance gap remains stable for most of the datasets when increasing the number of landmarks. This empirically verifies the efficiency of our repair strategy.

6.3 Scalability Test

We conducted a scalability test on the update time of our method IncHL$^+$, by starting with 500 updates and then iteratively adding 500 updates each time until 10,000 updates. Figure 4 shows the results. We observe that the update time of IncHL$^+$ on almost all the datasets is considerably below the construction time of labelling. On Indochina and IT, IncHL$^+$ performs relatively worse because these networks have large average distances as depicted in Table 2, which lead to high percentages of affected vertices as shown in Figure 1. In contrast, IncHL$^+$ performs well on graphs with small average distances such as Twitter. Overall, IncHL$^+$ can scale to perform a large number of updates efficiently.

7 CONCLUSION

This paper has studied the problem of answering distance queries on large dynamic networks. Our proposed algorithm exploits properties of a recent labelling technique called highway cover labelling [10] to efficiently process incremental graph updates, and can preserve the minimality property of labelling after each update operation. We have empirically evaluated the efficiency and scalability of the proposed algorithm. The results show that our proposed algorithm outperforms the state-of-the-art methods. In future, we plan to further investigate the effects of decremental updates on graphs since they are also commonly used in practice.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*. 230–241.

[2] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2012. Hierarchical hub labelings for shortest paths. In *ESA*. 24–35.

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *ACM SIGMOD*. 349–360.

[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2014. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*. 237–248.

[5] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *ACM SIGKDD*. 44–54.

[6] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. 2006. Complex networks: Structure and dynamics. *Physics reports* 424, 4-5 (2006), 175–308.

[7] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW*. 595–601.

[8] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[9] Gianlorenzo D'angelo, Mattia D'emidio, and Daniele Frigioni. 2019. Fully Dynamic 2-Hop Cover Labeling. *JEA* 24, 1 (2019), 1–6.

[10] Muhammad Farhan, Qing Wang, Yu Lin, and Brendan McKay. 2019. A Highly Scalable Labelling Approach for Exact Distance Queries in Complex Networks. In *EDBT*. 13–24.

[11] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. 2013. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *VLDB* 6, 6 (2013), 457–468.

[12] Takanori Hayashi, Takuya Akiba, and Ken-ichi Kawarabayashi. 2016. Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks. In *CIKM*. 1533–1542.

[13] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. 2012. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *ACM SIGMOD*. 445–456.

[14] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *WWW*. 1343–1350.

[15] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM TKDD* 1, 1 (2007), 2–es.

[16] Jure Leskovec and Andrej Krevl. 2015. SNAP Datasets:Stanford Large Network Dataset Collection. (2015).

[17] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *AAAI*.

[18] Robert Endre Tarjan. 1983. *Data structures and network algorithms*. Vol. 44. Siam.

[19] Antti Ukkonen, Carlos Castillo, Debora Donato, and Aristides Gionis. 2008. Searching the wikipedia with contextual information. In *CIKM*. 1351–1352.

[20] Monique V Vieira, Bruno M Fonseca, Rodrigo Damazio, Paulo B Golgher, Davi de Castro Reis, and Berthier Ribeiro-Neto. 2007. Efficient search ranking in social networks. In *CIKM*. 563–572.

[21] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P Gummadi. 2009. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*. 37–42.

[22] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *ACM SIGMOD*. 99–110.

Short Paper

KISS — A fast kNN-based Importance Score for Subspaces

Anna Beer
LMU Munich
Munich, Germany
beer@dbs.ifi.lmu.de

Ekaterina Allerborn
LMU Munich
Munich, Germany

Valentin Hartmann
EPFL
Lausanne, Switzerland
valentin.hartmann@epfl.ch

Thomas Seidl
LMU Munich
Munich, Germany
seidl@dbs.ifi.lmu.de

ABSTRACT

In high-dimensional datasets some dimensions or attributes can be more important than others. Whereas most algorithms neglect one or more dimensions for all points of a dataset or at least for all points of a certain cluster together, our method KISS (***k***NN-based **I**mportance **S**core of **S**ubspaces) detects the most important dimensions for each point individually. It is fully unsupervised and does not depend on distorted multidimensional distance measures. Instead, the k nearest neighbors (kNN) in one-dimensional projections of the data points are used to calculate the score for every dimension's importance. Experiments across a variety of settings show that those scores reflect well the structure of the data. KISS can be used for subspace clustering. What sets it apart from other methods for this task is its runtime, which is linear in the number of dimensions and $O(n \log(n))$ in the number of points, as opposed to quadratic or even exponential runtimes for previous algorithms.

1 INTRODUCTION

As sensors in fields like biology and chemistry become more and more sophisticated, websites collect more and more data about their users, and IoT and manufacturing devices get equipped with sensors that allow for predictive maintenance, the amount, granularity and dimensionality of data increases. In order to still be able to analyze the data in a meaningful way and in a reasonable time, one often needs to reduce at least one of the three; this paper will focus on the dimensionality. The more dimensions there are, the more of them are not important and distort further data mining tasks. The more dimensions, the longer it takes to process them all: the running time of many algorithms increases exponentially with the number of dimensions, especially of those designed for fewer dimensions. But not only that: many distance measures become more and more useless with an increasing number of dimensions [4]. Thus, instead of dragging along all dimensions of a point, many methods focus on working on only a small subset of the dimensions. The dimensions that a point is reduced to should, of course, be the ones that capture the most relevant information that this point contains. But to learn those important dimensions proves difficult: users do not want to waste time studying the data and its features thoroughly before applying a data mining algorithm. However, most algorithms still require user input that needs expert knowledge, or even require the user to label data by hand. In addition, the number of possibly important subspaces is exponential, making it a complex and time-consuming task to find the most important one. As datasets grow in size and contain data from different sources, one part of a dataset might differ a lot from a different part. Nonetheless, methods for dimensionality reduction typically try to find a common subspace for all data points, which can potentially be completely unsuited for heterogeneous data. Often, every single point has its own properties and thus the importance of a subspace may vary for each point, as shown in Fig. 1: for objects 1,4,7 in the first column the texture may be relevant, whereas it does not seem to be important for the other objects, since every other texture only occurs once. Also, for objects 1,2,3 in the first row and the quadrangles 4 and 5 in the second row the number of corners may be important. The color could be the best attribute to distinguish objects 1,5, and 9 in the diagonal from the others. Thus, for object 1 all three considered dimensions — number of corners, color, and texture — may be relevant, while there are other objects in the same dataset for which not all of those dimensions are important, e.g., for object 7 only the texture is relevant.

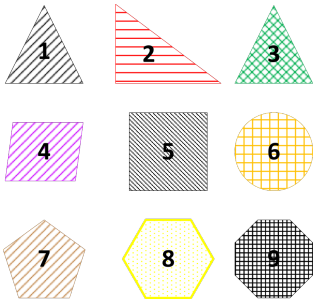


Figure 1: For different objects, different attributes or subspaces can be relevant: texture, number of corners, color, or a subset of those dimensions may be important.

A method to score the importance and expressiveness of each dimension for every point of a dataset individually without requiring any user input that scales to high dimensionalities would solve the problems mentioned above. In this paper, we develop KISS, a ***k***NN-based **I**mportance **S**core of **S**ubspaces, which fulfills all of these requirements. KISS can detect the most important subspace for a point fast and reliably in highly noisy data and data where only few dimensions are important per point.

One of the fundamental considerations that led to KISS is that those dimensions are most expressive for a point whose values lie in a cluster. We use the observation that if a point lies in a cluster in a certain subspace, the kNN of the projections of this point onto each dimension of this subspace intersect heavily. Since kNN in one-dimensional projections of the data can be computed fast, we

can efficiently calculate a score indicating the likelihood of the point lying in a cluster in the considered dimension. Usage of the kNN prevents relying on non-expressive distance measures, and there is no need for the users to know the data beforehand. KISS is deterministic, simple, fully unsupervised, and scalable w.r.t. the number of points as well as to the number of dimensions. It is easy to implement and reliably detects important dimensions for individual points fast. Our main contributions are as follows:

- We develop KISS, an importance scoring for every dimension for each individual point.
- KISS is fast w.r.t. both the number of points as well as the number of dimensions.
- KISS is fully unsupervised
- KISS does not rely on any multidimensional distance measure that gets useless for a high number of dimensions.

2 RELATED WORK

The problem of finding a global important subspace for all points has been addressed in previous work, which we introduce in Section 2.1. We restrict ourselves to algorithms that, like KISS (and contrary to, e.g., PCA or FOSSCLU), work in the standard basis of the vector space, as it simplifies getting insights into the data, which KISS was developed for.

2.1 Subspace Search

There exists some work on scoring of dimensions, where RIS, SURFING, and SCHISM are some of the most common algorithms.

RIS [6] produces a ranked list of all dimensions using a density-based quality criterion ("interestingness") that requires multiple parameters, which are set based on heuristic methods. The rating is only a relative comparison between different dimensions of the same dataset and is the same for all points.

SURFING [3] is a bottom-up approach that also returns the most "interesting" subspaces of a dataset. It is, like KISS, based on kNN, declaring subspaces as interesting in which "the k-nn-distances of the objects differ significantly from each other" [3]. The kNN distances are computed w.r.t. the subspaces, making their expressiveness dependent on the dimensionality of the subspaces. The algorithm has a runtime complexity of $O(mn^2)$, where n is the number of points and m is the number of different subspaces analyzed, which is 2^D in the worst case, making it much less scalable regarding both the number of points as well as the number of dimensions. Additionally, the minimum cluster size k has to be specified by the user.

SCHISM [8] extends the CLIQUE [2] principle and looks at the density of grid cells using an adaptive threshold function τ given by the user and applying the Chernoff-Hoeffding bound. It uses several preprocessing steps and requires three user given parameters u, τ, and ξ. Like RIS, and in contrast to KISS, it calculates a global score for "interesting" subspaces that is not adapted to individual points.

Although the dimension weightings at first glance seem to be suitable for comparing with KISS, such a comparison proves difficult: These dimension scoring methods do not return important subspaces for each point individually, or require at least two parameters set by the user, making it hard to objectively evaluate without overoptimism. But most notably, they are far more complex: The fastest of them, RIS, has runtime at least quadratic in the number of dimensions as well as the number of points. SCHISM is only linear in the number of points, but exponential in the number of dimensions. SURFING is quadratic in the number of points and exponential in the number of dimensions.

2.2 Subspace Clustering

We do not perform any clustering in this paper, but since we define "important dimensions" as dimensions in which a point lies in a cluster, there is a relation to the field of subspace clustering. Even though subspace clustering algorithms also deliver important dimensions in a way, their focus is different from KISS. Whereas those algorithms often need to perform a complete clustering of the dataset, we aim to get the relevant subspaces directly, individually for every point. We do not need to know the precise clusters to find important dimensions. Also, most of those algorithms rely on parameters that are not easy to set. We found that especially our first two goals, being fully unsupervised, and returning individual scores for different points, are to the best of our knowledge not achieved simultaneously by any other algorithm in this field. COSA and DISH are most related to our work, since they both consider subspaces for individual points:

COSA [5] finds important subspaces individually for each point using the kNN. A hierarchical clustering is applied based on a dimension weighting matrix and the relevant dimensions can be calculated based on the dimension weights of the respective cluster members. Despite the similarities to KISS, there are two major differences: First, users need to set a not quite intuitive parameter λ, which gives the "strength of incentive for clustering on more dimensions" [7]. Second, the kNN are calculated in the full-dimensional space, making COSA vulnerable to the loss of expressiveness of distance measures in high dimensions.

DiSH [1] is a density-based algorithm that finds cluster hierarchies and nested clusters. It has two parameters: a smoothing factor μ representing the minimum number of points in a cluster and ε for ε-range queries. Even though DiSH also uses only one-dimensional range queries and delivers subspace preference vectors for every point, the nesting of subspaces makes it impossible to determine the distinctly important subspaces. Also, the vectors are only calculated in an intermediate step, and depend on the parameter choices.

2.3 Possible Competitors

Finding suitable methods to compare KISS to is difficult: there are a number of subspace clustering algorithms, but they perform clustering, and not detection of the most important subspaces. For some algorithms one could extract the important subspaces of a point by looking at the subspace of the cluster the point was assigned to. This assignment, however, can only be obtained after an expensive clustering of the complete dataset. Some algorithms like, e.g., RIS or SURFING rank the subspaces in a similar way as KISS scores them, but they deliver one ranking for the complete dataset, not for each point individually. To the best of our knowledge, there is no algorithm yet that fulfills all of the requirements we impose. In particular, returning individual dimension ratings for each point and being completely unsupervised are very rare properties. Nevertheless, to at least have *some* point of reference, we exemplarily compare against CLIQUE, which is a grid-based bottom-up approach for subspace clustering. It requires two parameters, ξ and τ, which determine the number of intervals every dimension is partitioned into and the density threshold. We try out different parameter settings, showing that the results are very

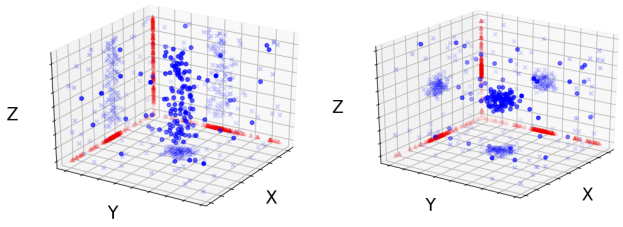


Figure 2: Projections of a 2-dimensional resp. 3-dimensional cluster. Blue crosses are projections onto the 2-dimensional subspace, red triangles projections onto one dimension.

sensitive to the parameter choices, whereas for KISS no parameters need to be tuned. In addition, we use the quality criterion of SURFING to obtain scores for every dimension and compare KISS to them. However, those are global scores for all points and not individual ones like those computed by KISS.

3 KISS

This section presents our newly developed dimension score KISS. We first describe the basic idea to use the k nearest neighbors in one-dimensional projections of the n data points to be able to compute KISS, which gives a scoring for the importance of every dimension for each individual point. Section 3.2 motivates our idea mathematically. In Section 3.3 we develop the exact formula of KISS, and present the complete KISS-based algorithm to obtain the most important subspace of a point. We analyze the complexity of our algorithm in Section 3.4.

3.1 Idea: Using One-dimensional kNN

If a point lies in a cluster in a d-dimensional subspace, most of the kNN of this point in the projection of the dataset onto those d dimensions will be members of that cluster, too. If we look at only one of those d dimensions (cf. the red triangles in Fig. 2), we still see the cluster structure: the cluster on the left lies in dimensions X and Y, and the red triangles on the according axes show a clear cluster structure. Projected onto those axes, most of the one-dimensional kNN of a point will lie in the same (original) d-dimensional cluster as the point itself.

Following this observation, for a given point p, we count for every other point q in how many dimensions it belongs to the one-dimensional kNN of p, giving us a Point Score $PS(p, q)$. A high Point Score means that q is likely to be contained in the same (higher-dimensional) cluster as p, meaning that the dimensions that they share carry more importance for p than the others. Summing up the Point Scores for each dimension individually gives us a measure for the importance of each dimension, where we account for outliers by incorporating the one-dimensional distances to the kNN of a point.

3.2 Mathematical Perspective

In the following we give some theoretical insights which support our idea. We denote cluster indices by superscripts and dimension indices by subscripts, and see clusters as collections of points drawn from a common probability distribution over $\mathbb{R}^D$.

Consider a cluster C^1 with center c^1 in dimensions $\{1, \dots, l\}$ (without loss of generality). We will consider the neighborhood of points in dimension 1. Let C^2 be a different cluster with center c^2 that overlaps with cluster C^1 in dimension 1, and assume that for all points $r^1 \in C^1$ and all $r^2 \in C^2$ we have $\Pr(|r_1^1 - c_1| \leq \varepsilon) \geq \delta$ and $\Pr(|r_1^2 - c_1| \leq \varepsilon) \geq \delta$, respectively, where $\varepsilon > 0$ and δ is a value close to 1. Furthermore, C^1 and C^2 should not overlap and be sufficiently far apart in the dimension that they share: $|c_1^1 - c_1^2| \leq 4\varepsilon + \varepsilon'$ for an arbitrarily small $\varepsilon' > 0$. This is, e.g., the case for all shared dimensions with high probability if c^1 and c^2 are uniform samples from a sufficiently large set in $\mathbb{R}^D$.
Let $p^1, \tilde{p}^1 \in C^1$, and $p^2 \in C^2$. In addition, let q be a point that does not lie in any cluster in dimension 1 and is drawn from a somewhat uniform distribution on a large enough interval. Precisely, we require it to fulfill $\Pr(|q - c_{1,2}| \leq 3\varepsilon) \leq \delta'$, where δ' is close to 0.
We now consider the distance of p^1 to the other three points in dimension 1.
For the point from the same cluster, we get

$$\Pr(|p_1^1 - \tilde{p}_1^1| \leq 2\varepsilon) \geq \Pr(|p_1^1 - c_1^1| + |c_1^1 - \tilde{p}_1^1| \leq 2\varepsilon)$$
$$\geq \Pr(|p_1^1 - c_1^1| \leq \varepsilon) \Pr(|c_1^1 - \tilde{p}_1^1| \leq \varepsilon) \geq \delta^2 \approx 1.$$

The point from the other cluster yields

$$\Pr(|p_1^1 - p_1^2| \leq 2\varepsilon) \leq \Pr(|p_1^1 - c_1^2| > \varepsilon \text{ or } |p_1^2 - c_1^1| > \varepsilon)$$
$$\leq \Pr(|p_1^1 - c_1^2| > \varepsilon) + \Pr(|p_1^2 - c_1^1| > \varepsilon) = 2(1 - \delta) \approx 0,$$

where for the first step we observed that at least one of p_1^1, p_1^2 needs to lie outside of the ε-interval around its cluster's center, and applied a simple union bound to obtain the second inequality. Finally, for the point that does not lie in any cluster in dimension 1, we have, using the same arguments as above,

$$\Pr(|p_1^1 - q_1| \leq 2\varepsilon) \leq \Pr(|p_1^1 - c_1^1| > \varepsilon \text{ or } |q_1 - c_1^1| \leq 3\varepsilon)$$
$$\leq \Pr(|p_1^1 - c_1^1| > \varepsilon) + \Pr(|q_1 - c_1^1| \leq 3\varepsilon) \leq (1 - \delta) + \delta' \approx 0.$$

Thus, if k is chosen smaller than the size of C^1, the kNN of p_1^1 will almost exclusively consist of points from C^1. Thus,

$$\mathbb{E}(|\{i \in \{1, \dots, l\} \mid \tilde{p}_i^1 \text{ is part of the } k\text{-NN of } p_i^1\}|) \approx l\frac{k}{|C^1|} > l\frac{k}{n},$$

where the last quantity corresponds to a uniform distribution of points.

3.3 The Full Algorithm

KISS, the score indicating the importance of a dimension d for a point p in a dataset DB, depends on the k nearest neighbors (kNN) of p in the one-dimensional projection onto d: $kNN_p(d)$. Note that their number can be larger than k in case of ties, since we chose the deterministic variant of kNN.

The more often a point occurs in the sets of one-dimensional nearest neighbors of p, the closer related it is to p, which we capture in the *Point Score* $PS(p, q)$, where $\mathbb{1}$ is the indicator function: $PS(p, q) = \sum_{d=1}^{D} \mathbb{1}\{q \in kNN_p(d)\}$.

Higher values for k lead to more accurate scores, as shown in Fig. 3. However, the runtime of our algorithm depends on k and we would like to keep it $O(n \log(n))$ in the number of points (see our complexity analysis in Section 3.4). For this reason, k is set to $\sqrt{n}$, which is also in line with previous literature [5].

The importance a dimension d has for a point p depends not only on the intersection of the kNN in this dimension with the kNN in the other dimensions, but also on the distance of those kNN. Otherwise, the important dimensions for outliers would be distorted. Thus, the farther away a point in the kNN is, the less influence it should have on the importance of the respective dimension, which is why we divide the Point Score of each

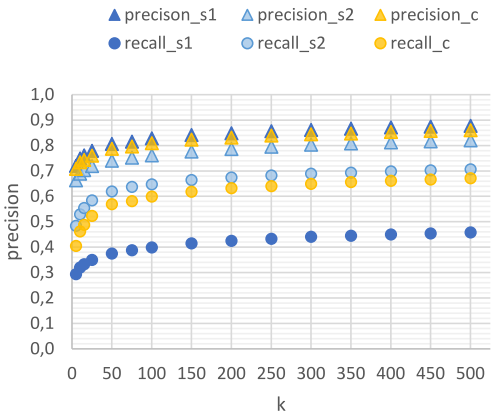


Figure 3: Results for different values of k. _s1, _s2, and _c denote different binarization methods, see Sec. 4.

$q \in kNN_p(d)$ by the distance between the corresponding projections of q and p. Additionally, the computed value is divided by the neighborhood size to account for ties among the nearest neighbors:

$$KISS'(p,d) = \frac{1}{|kNN_p(d)|} \sum_{q \in kNN_p(d)} \frac{1}{dist(p_d, q_d)} PS(p,q) \quad (1)$$

Finally, KISS is normalized for every point by dividing every value by the highest KISS occurring for the respective point:

$$KISS(p,d) = \frac{KISS'(p,d)}{\max_{e \in \{1,\ldots,D\}} KISS'(p,e)}. \quad (2)$$

This gives a value between 0 and 1 and allows for a meaningful comparison between different points of a dataset.

3.4 Complexity

The calculation of the kNN of all points in all dimensions needs $O(D * k * n + D * n * \log(n))$ steps, where D is the number of dimensions in the data set DB of size $|DB| = n$. Computing the kNN in one dimension can be performed efficiently by sorting the points w.r.t. this dimension and going to the left and right of the query point in the sorted list. Given the kNN of every point for every dimension, all Point Scores $PS(p,\cdot)$ w.r.t. a point p can be calculated in $O(D*k)$ by iterating through the k nearest neighbors of p in all D dimensions, keeping track of the scores via a hashmap where they get continuously updated. This has to be done for all n points, resulting in $O(n * D * k)$. For calculating the KISS for a point p and a dimension d, we need to sum up the Point Scores of all of p's kNN in d divided by their (one-dimensional) distance in this dimension, which can be done in $O(1)$. The summation can be performed in $O(k)$. We want to compute the KISS for all points and all dimensions, thus we get $O(n * D * k)$.

The KISS for all dimensions and all points can hence be computed in time $O(D*k*n+D*n*\log(n)+n*D*k+n*D*k) = O(D*n*(k+\log(n))$, which is linear in the dimension D and close to linear in the size of the dataset n. Runtime experiments confirmed this behavior, but were omitted due to space constraints.

4 EXPERIMENTAL EVALUATION

In Section 4.1 we introduce a technical tool that is needed to validate our method against a ground truth. In Section 4.2 we describe our experiments, and summarize the results in Section 4.3.

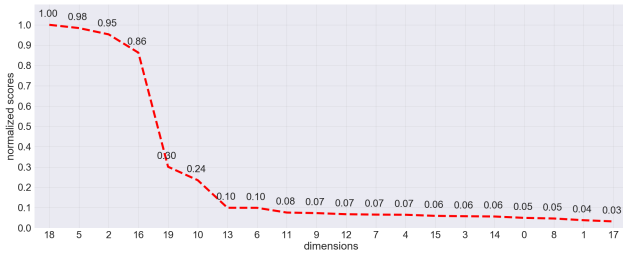


Figure 4: Typically distributed scores for different dimensions for a point p, sorted by descending normalized score.

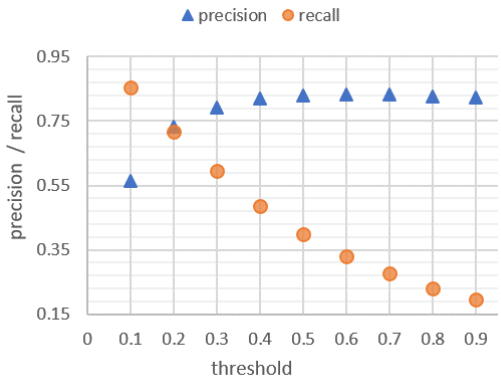


Figure 5: Precision and recall using simple binarization with different thresholds.

4.1 Binarization

To be able to validate our results and because for certain applications a division of the dimensions into important and unimportant ones can be needed, we suggest two possibilities to binarize the values obtained by KISS. Ordered by score value, a typical distribution of the scores for a point is shown in Fig. 4. If a point lies in a cluster, the KISS of the according dimensions clearly differs from the KISS of unimportant dimensions.

The naïve approach "simple binarization" of using a fixed threshold for the normalized score based on which we set the score to either 0 or 1, already delivers good results, as we show in Section 4.2. Fig. 5 shows recall and precision for our base case experiment and different values for the threshold, where 0.2 offers a good trade-off between the two. We performed the other experiments with the thresholds 0.5 and 0.2, denoted by _s1 and _s2, respectively.

Additionally, we developed a more sophisticated approach — "complex binarization" —, which comes with an only negligable increase in runtime, to improve our results even further. Here, we look for the most appropriate cut position in the ranked scores: e.g., for the KISS distribution depicted in Fig. 4 it could be dimension 12 since the scores for all dimensions to the right of it are significantly lower than the ones to the left of it. Our approach for detecting this cut position in the score ranking consists of first setting the importance of each dimension to 1 and then lowering it to 0 if its KISS lies below one of the three thresholds described below.

We set all parameters required for the complex binarization to the same reasonable values we give below for all experiments. Both strategies are introduced mainly to be able to validate our results against a binary ground truth. Note that the parameter values rely on the data being scaled to the D-dimensional unit

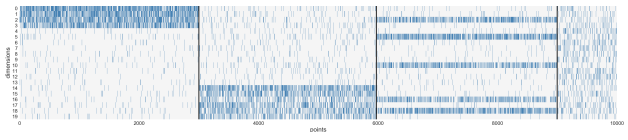


Figure 6: Transpose of the binary matrix for the base case dataset computed using KISS with complex binarization. The subspace boundaries are depicted as black lines.

hypercube. The three thresholds for the complex binarization are:

(1) *normalized threshold* t_n: If $KISS(p,d) < t_v = 0.1$, then $KISS(p,d)$ is set to 0.
(2) *unnormalized threshold* t_u: Because we normalize the KISS of each dimension by dividing by the largest KISS for this point, even a point that is just random noise has at least one dimension with score 1. However, the unnormalized value of dimensions with a high normalized KISS for this noise point will be significantly lower than the unnormalized values of dimensions with a high normlized KISS for a point that lies in a cluster. Thus, in addition to setting a threshold for the normalized KISS, we also set one for the unnormalized KISS' (cf. Equation 1): if $KISS'(p,d) < t_u$, the score for d is set to 0, where t_u equals the difference between mean and minimum of all unnormalized scores.
(3) *descent threshold* t_d: The descent threshold controls the decline between consecutive KISS values. If $\frac{KISS(p,e)-KISS(p,d)}{KISS(p,e)} > t_d = 0.7$, where $KISS(p,e)$ is the next largest KISS value of p, then all KISS values smaller than or equal to $KISS(p,d)$ become 0.

The result of the binarization can be expressed in a binary matrix as in Fig. 6.

Empirically, setting t_n significantly lower than 0.5 and t_d rather high allows for detecting more relevant dimensions and therefore detecting subspaces of higher dimensionality. t_u affects mostly how well outliers are detected. However, setting this parameter too high leads to very restricted binarized scores and can possibly decrease the detection rate of the important dimensions of the cluster points. In general, the parameters allow us to trade precision for recall. KISS is supposed to be used in settings where working with the original data without a significant reduction of the dimensionality is infeasible, either due to limited human capacity when manually analyzing the data or due to non-favorable dependence of a downstream task's performance on the number of dimensions. Hence we can live with a mediocre recall if in return the precision is high, allowing us to get rid of many dimensions, which is why we mainly focus on achieving a high precision.

In real-world settings where one needs a binary division of the dimensions, one typically has a (computational or storage) budget of dimensions one can deal with in the downstream task, and would binarize in a way so that exactly this many dimensions are labeled as important.

4.2 Experiments

We test with both the simple and complex binarization of KISS and denote the corresponding values with the abbreviations _s and _c, respectively. To have ground truth values for the importance of all dimensions, we generated data containing subspace clusters with possibly overlapping subspaces. The clusters are

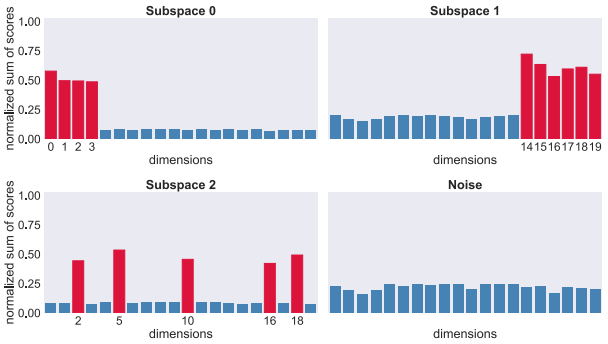


Figure 7: Average KISS for all points, partitioned according to ground-truth based important subspaces. Red bars show dimensions containing clusters.

Gaussians with mean randomly drawn from the uniform distribution on the D-dimensional unit hypercube. The values for the dimensions of a point that do not lie in a cluster are uniformly distributed in the hypercube [1].

Looking at the distribution of KISS per important (according to the ground truth) subspace in Fig. 7, we already see the correlation to the cluster subspaces: the average scores for the important dimensions (red bars) are visibly higher than those for unimportant dimensions (blue bars). Noise points that do not lie in any cluster are shown in the lower right diagram: the average KISS values do not differ much. The precision we achieve for both types of binarization are good, as can be seen in Fig. 8.

To the best of our knowledge there are no alternatives yet to KISS (see Section 2.3). However, with CLIQUE and SURFING we compare KISS to representative algorithms for subspace clustering and for subspace search. The comparison makes the disadvantages of having to set parameters as well as the benefit of individual scores in contrast to a global ranking clear.

Among others, Fig. 8 shows results obtained with CLIQUE for different parameter configurations. Even though CLIQUE was able to obtain high recall values, the precision was even for the best parameter settings much lower than for KISS (for both binarization methods). We also see that the results heavily depend on the choice of CLIQUE's parameters, with precision ranging from 35% to 77% and recall from 53% to 87%. Fig. 8 further includes the results obtained by binarizing the "quality" of each particular dimension as computed by SURFING for different values of k (in the same way as we binarize the KISS values). The classification performance of SURFING is very dependent on the parameter choice as well. With a good choice, it is able to achieve a high recall, but, as expected, the precision values are rather low, since the quality assignments are the same for all points, which does not match the ground truth.

Starting from this base case, we altered one parameter of the data in each of the following subsections to investigate KISS' behaviour w.r.t. this parameter.

4.2.1 Number of points. Changing the number of points n did not affect the precision, recall or accuracy of KISS significantly.

[1] The settings for our base case dataset are as follows: number of points $n = 10000$, dimensionality of point p: $dim(p) = 20$, percentage of noise $noise = 0.1$, set of dimensions in subspace S_i: $S_0 = \{0 \dots 3\}$, $S_1 = \{14 \dots 19\}$, $S_2 = \{2, 5, 10, 16, 18\}$, percentage of points w.r.t. n lying in subspace S_i: $|S_i| = [0.3, 0.3, 0.3]$, dimensionality of S_i: $dim(S_i) = [4, 6, 5]$, number of clusters in subspace S_i: $n_c(S_i) = [1, 2, 1]$, variance of cluster C_i: $var(C_i) = [1.5, 1.0, 1.3]$.

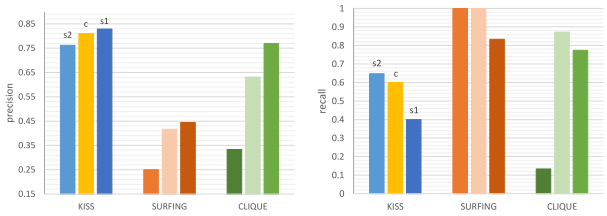


Figure 8: Results for KISS, SURFING, and CLIQUE for the base case dataset and different parameters ξ and τ resp. k. Same color indicates same parameters.

The three values deviated by at most 3% for $n \in \{5\,000, 10\,000, 25\,000, 50\,000, 75\,000, 100\,000\}$.

4.2.2 Number of dimensions. With growing number of dimensions (while keeping the ratio of dimensions lying in important subspaces the same) the recall decreases, but the precision, which we put more emphasis on, stays high.

4.2.3 Noise. Increasing the percentage of pure noise points leads to less points in each cluster, thus precision drops. Nevertheless, the decrease of quality is slow, and up to 50% of data can be pure noise points before precision falls below 75% (for the complex binarization).

4.2.4 Number of subspaces. We examine KISS for up to 10 different subspaces and obtain good results with precisions above 73% in all cases. Additionally, recall as well as accuracy diminish only slightly with increasing number of subspaces.

4.2.5 Subspace size ratio. We tested several size ratios between the three base case subspaces our dataset consists of. In our base case, every subspace contains one third of the non-noise data points. We set the share of instances in the first subspace to values $\{0.4, 0.5, 0.6, 0.7, 0.8\}$, while dividing the remaining points equally among the other two subspaces (and additionally keeping the 10% noise of the base case). The quality of the scores hardly changed: we received precision values for the simple binarization between 83% and 85%, and between 84% and 86% for the complex binarization. Recall values ranged between 39% and 41%, and 47% and 49%, respectively, showing that the size ratio of the subspaces does not constitute a problem for KISS. Thus, even subspaces containing only very few points of the complete dataset can be found as easily as bigger subspaces.

4.2.6 Number of clusters per subspace. With an increasing number of clusters, precision as well as recall decrease, since there are fewer points per cluster that could help identify a point in the cluster.

4.2.7 Density of clusters. We tested several density settings for the clusters. When all subspaces contain similarly dense clusters, the quality decreases with lower density (i.e., higher standard deviation). If each subspace contains differently dense clusters, the results are rather determined by the average density than by the lowest or highest occurring density. Thus, a large difference in cluster density does not influence the results negatively.

[1]When adding more subspaces to the base case dataset, we use the same settings as for the original subspaces: the points are evenly distributed among the subspaces, and the cluster settings of the clusters lying in subspaces S_{0+3i}, S_{1+3i}, S_{2+3i} correspond to the settings of the clusters lying in subspaces S_0, S_1, S_2.

4.3 Summary of Results

Our experiments show that KISS achieves a high precision and reasonable recall across a wide range of settings. With a high number of subspaces or clusters the performance starts to degrade, but KISS is robust to noise and can deal with high numbers of points as well as clusters of different density. We would like to point out that the experiments only show a small part of KISS' capabilities, since the original KISS is a continuous value, which we just binarized here, and likely not even optimally.

5 CONCLUSION AND FUTURE WORK

We developed KISS, a scoring that assigns an importance value to each dimension of each point of a dataset. It is scalable, does not suffer from the curse of dimensionality, since it replaces multi-dimensional distance measures by one-dimensional ones, and does not require significant user involvement to set parameters. Its runtime is linear in the dimensionality and close to linear in the number of points, setting it apart from similar methods.

KISS has numerous applications, both as a tool to get an insight into datasets as well as a foundation for data mining applications, in particular to accelerate downstream tasks or to make them more robust to noise. We are currently working on some of the most immediate extensions: (1) performing clustering using especially the most relevant dimensions for each point; and (2), using KISS for outlier and noise detection, following the observation that points that have a low KISS in every dimension are typically in none of those in a cluster. We encourage the usage of KISS for preprocessing data and gaining knowledge in an early stage of a data anlysis process, since it is simple, fast, delivers good results and does not require parameter tuning.

ACKNOWLEDGMENTS

This work has been partially funded by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IS18036A. The authors of this work take full responsibilities for its content.

REFERENCES

[1] Elke Achtert, Christian Böhm, Hans-Peter Kriegel, Peer Kröger, Ina Müller-Gorman, and Arthur Zimek. 2007. Detection and visualization of subspace cluster hierarchies. In *International Conference on Database Systems for Advanced Applications*. Springer, 152–163.

[2] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. 1998. *Automatic subspace clustering of high dimensional data for data mining applications*. Vol. 27. ACM.

[3] Christian Baumgartner, Claudia Plant, K Railing, H-P Kriegel, and Peer Kroger. 2004. Subspace selection for clustering high-dimensional data. In *Data Mining, 2004. ICDM'04. Fourth IEEE International Conference on*. IEEE, 11–18.

[4] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1999. When is "nearest neighbor" meaningful?. In *International conference on database theory*. Springer, 217–235.

[5] Jerome H Friedman and Jacqueline J Meulman. 2004. Clustering objects on subsets of attributes (with discussion). *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 66, 4 (2004), 815–849.

[6] Karin Kailing, Hans-Peter Kriegel, Peer Kroeger, and Stefanie Wanka. 2003. Ranking interesting subspaces for clustering high dimensional data. In *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 241–252.

[7] Lance Parsons, Ehtesham Haque, and Huan Liu. 2004. Subspace clustering for high dimensional data: a review. *Acm Sigkdd Explorations Newsletter* 6, 1 (2004), 90–105.

[8] Karlton Sequeira and Mohammed Zaki. 2004. SCHISM: A new approach for interesting subspace mining. In *Fourth IEEE International Conference on Data Mining (ICDM'04)*. IEEE, 186–193.

Short Paper

SceneRec: Scene-Based Graph Neural Networks for Recommender Systems

Gang Wang
SKLSDE Lab, Beihang University
iegwang@buaa.edu.cn

Ziyi Guo
JD.com
guoziyi@jd.com

Xiang Li
East China Normal University
lixiang3776@gmail.com

Dawei Yin
Baidu.com
yindawei@acm.org

Shuai Ma
SKLSDE Lab, Beihang University
mashuai@buaa.edu.cn

ABSTRACT

Collaborative filtering has been largely used to advance modern recommender systems to predict user preference. A key component in collaborative filtering is representation learning, which aims to project users and items into a low dimensional space to capture collaborative signals. However, the scene information, which has effectively guided many recommendation tasks, is rarely considered in existing collaborative filtering methods. To bridge this gap, we focus on scene-based collaborative recommendation and propose a novel representation model SceneRec. SceneRec formally defines a *scene* as a set of pre-defined item categories that occur simultaneously in real-life situations and creatively designs an item-category-scene hierarchical structure to build a scene-based graph. In the scene-based graph, we adopt graph neural networks to learn scene-specific representation on each item node, which is further aggregated with latent representation learned from collaborative interactions to make recommendations. We perform extensive experiments on real-world E-commerce datasets and the results demonstrate the effectiveness of the proposed method.

1 INTRODUCTION

Recommender systems have become increasingly important to address the information overload problem and have been widely applied in many different fields, such as social networks [22] and news websites [24]. To predict a user's preference, an extensive amount of collaborative filtering (CF) methods have been proposed to advance recommender systems. The basic idea of CF is that user behavior would always be similar and a user's interest can be predicted from the historical interactive data like clicks or purchases. A key component of CF is to learn the latent representation, which usually projects users and items into a lower dimensional space. A variety of CF models, including matrix factorization [8], deep neural networks [7] and graph convolutional networks [16], are adopted to capture collaborative signals from a user-item matrix or a user-item bipartite graph.

In the meantime, recommender systems that integrate scene information are attracting more and more attention. For example, predictive models are able to recommend substitutable or complementary items [9, 10, 13] that visually match the scene which is represented in an input image. The image data contains rich contextual information like background color, location, landscape, etc., which may be ignored by conventional CF methods. However, the input image could reveal no scene information or even becomes unavailable in many recommendation scenarios. For example, in E-commerce systems, most thumbnail images only contain product pictures which are embedded in the white background. In such circumstances, scene-based recommendation becomes infeasible because the scene definition is not clear.

To address this issue, this work investigates the utility of incorporating scene information into CF recommendation. However, this study brings two challenges. First, a formal definition on scene is essential to this problem. Without image data, how to formally define a scene becomes a problem. Second, how to incorporate scene information into existing CF models should also be taken into account. Keeping these two key points in mind, we propose SceneRec, a novel method for scene-based collaborative filtering. Specifically, we propose a principled item-category-scene hierarchical structure to construct the scene-based graph (Figure 1). In particular, a *scene* is formally defined by a set of fine-grained item categories that could simultaneously occur in real-life situations. For example, the set of item categories {Keyboard, Mouse, Mouse Pad, Battery Charger, Headset} represents the scene "Peripheral Devices". This can be naturally applied to a situation where a user has already bought a PC and many different types of supplementary devices are recommended. Moreover, SceneRec applies graph neural networks on the scene-based graph to learn the item representation based on the scene information, which is further aggregated with the latent representation learned from user-item interactions to make predictions.

To the best of our knowledge, SceneRec is among the first to study scene-based recommendation with a principled scene definition and our main contributions are summarized as follows:
(1) We study the problem of scene-based collaborative filtering for recommender system where a scene is formally defined as a set of item categories that could reflect a real-world situation.
(2)We propose a novel recommendation model SceneRec. It leverages graph neural networks to propagate scene information and learn the scene-specific representation for each item. This representation is further incorporated with a latent representation from user-item collaborative interactions to make predictions.
(3) We conduct extensive experiments to evaluate the performance of SceneRec against 9 other baseline methods. We find that our method SceneRec is effective. Specifically, SceneRec on average improves the two metrics (NDCG@10, HR@10) over the baselines by (14.8%, 12.1%) on 4 real-world datasets.

2 RELATED WORK

Collaborative filtering has been widely applied in modern recommender systems. One class of CF methods try to build explicit models on the user-item interactions. For example, matrix factorization [2, 8, 12, 14] maps the representation of each user and each item into a lower dimensional space and calculates inner product

between vector representations to make predictions. To enhance recommendation, various contextual information has been incorporated into CF, such as user review [21], social connections [22] and item side information [17]. Different from existing works that rely on linear predictive function, many recent efforts apply deep learning techniques [7] to learn non-linearities between user embedding and item embedding.

Another line of CF methods take user-item interactions as a bipartite graph. For example, some early efforts [5] conduct label propagation, which essentially searches neighborhood on the graph, to capture collaborative signals. Inspired by the success of graph neural networks (GNN) [6, 11] that directly conduct convolutional operations on the non-grid network data, a series of GNN-based recommendation methods have been proposed on an item-item graph [23] or a user-item graph [16] to learn a vector embedding for each item or user. The general idea is the representation of one graph node can be aggregated and combined by the representation of its neighbor nodes. NGCF [20] extends GNN to multiple depths to capture high-order connectivities that are included in user-item interactions. KGAT [19] and KGCN [18] investigate the utility of incorporating knowledge graph (KG) into CF by projecting KG entities to item nodes.

Our work is also related to the application of scene information in recommender systems. For example, given the scene in the form of an input image, recommendation methods are capable of providing substitutable [10, 13] or supplementary[9] products that visually match the input scene. However, in these tasks, the scene is represented by image data, which is not readily available in many recommendation scenarios. In such cases, scene-based recommendations become difficult or even impossible because the scene has not been well defined. In this paper, we aim to integrate scene information into CF where each scene is define by a set of fine-grained item categories. By exploiting the scene-specific representation into conventional CF signals, the model can potentially improve predictions on user preference.

3 PROBLEM FORMULATION

Definition 3.1. **Scene**. A *scene* is defined as a set of item categories that occur simultaneously and frequently in a real-life situation, denoted as $s = \{c_1, c_2, \cdots, c_{|s|} | c_i \in C, 1 \leq i \leq |s|\}$, where C is the set of item categories and $|s| \geq 1$. The item category is one of a item's attributes and $s \subset C$.

Definition 3.2. **User-Item Bipartite Graph.** The user-item interactions can be represented as a bipartite graph $\mathcal{G} = \{(u, x_{ui}, i) | u \in \mathcal{U}, i \in \mathcal{I}\}$, where $\mathcal{U}$ and $\mathcal{I}$ are the set of users and items respectively, and the edge x_{ui} indicates the occurrence or frequency with that the user u has interacted with the item i, such as clicking and purchasing.

Definition 3.3. **Scene-based Graph.** The scene-based graph $\mathcal{H}$ is a hierarchical network with three layers: the item layer, the category layer, and the scene layer as shown in Figure 1. The item layer consists of items and is denoted as $\mathcal{L}_{item} = \{(i_p, y_{pq}, i_q) | i_p, i_q \in \mathcal{I}\}$, where the edge y_{pq} represents the similarity between two items i_p and i_q. The category layer is denoted as $\mathcal{L}_{cate} = \{(c_p, z_{pq}, c_q) | c_p, c_q \in C\}$, where the edge z_{pq} represents that the category c_p has relevance to the category c_q. The interaction between the item layer and the category layer is described by $\mathcal{L}_{ic} = \{(i_p, a_{pq}, c_q) | i_p \in \mathcal{I}, c_q \in C\}$, where the edge a_{pq} connects an item i_p to a pre-defined item category c_q. The scene layer is composed of scenes, where a scene s is formally defined as a set of item categories $\{c_1, c_2, \cdots, c_{|s|}\}$.

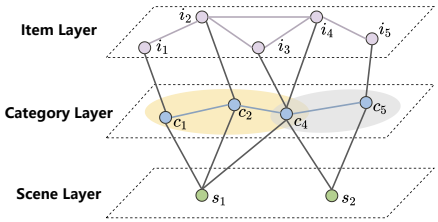


Figure 1: An illustrative example of the scene-based graph that consists of the item layer, the category layer and the scene layer. Each item is associated with a category. In the item layer and the category layer, the set of edges represent the item-item relations and the category-category relations. There are connections between categories and scenes, which indicates that a category belongs to a scene.

The relation between categories and scenes is illustrated by $\mathcal{L}_{cs} = \{(c_p, b_{pq}, s_q) | c_p \in C, s_q \in \mathcal{S}\}$, where the edge b_{pq} indicates that a category c_p belongs to a scene s_q and $\mathcal{S} = \{s_1, s_2, \cdots\}$ is the set of scenes. For simplicity, we set the weights of edges in the scene-based graph $\mathcal{H}$ to be 1; otherwise, 0.

Definition 3.4. **Scene-based Recommendation.** Given a user-item bipartite graph $\mathcal{G}$ recording interaction history, the goal of the scene-based recommendation is to predict the probability $\mathbf{r}_{ui}$ that the user u has potential interest in the item i with the help of scene information from a scene-based graph $\mathcal{H}$.

4 FRAMEWORK

In this section, we will first give an overview about the proposed framework, then introduce each model component in detail.

4.1 Architecture Overview

The architecture of the proposed model is shown in Figure 2. There are three components in the model: user modeling, item modeling, and rating prediction. User modeling aims to learn a latent representation for each user. To achieve this, we take user-item interaction as input and aggregate the latent representation of items that the user has interacted with to generate the user latent factor. Item modeling aims to generate the item latent factor representation. Since each item exists in both user-item bipartite graph and the scene-based graph, SceneRec learns item representations in each graph space, i.e., item modeling in the user-based space and item modeling in the scene-based space. In the user-based space, we take a similar strategy which aggregates the representation of all users that each item has interacted with to generate vector embedding. In the scene-based space, we exploit the hierarchical structure of the scene-based graph where the information is propagated from the scene layer to the category layer and from the category layer to the item layer. Then we concatenate two item latent factors for the general representation. In the last component, we integrate item and user representations to make rating prediction.

4.2 User Modeling

In the user-item graph, a user u_p is connected with a set of items and these items directly capture the user's interests. We thus learn user u_p's embedding $\mathbf{m}_{u_p}$ by aggregating the embeddings of item neighbors, which is formulated as,

$$\mathbf{m}_{u_p} = \sigma(\mathbf{W}_{\mathbf{u}} \cdot \left\{ \sum_{i_q \in UI(u_p)} \mathbf{e}_{i_q} \right\} + \mathbf{b}_{\mathbf{u}}), \tag{1}$$

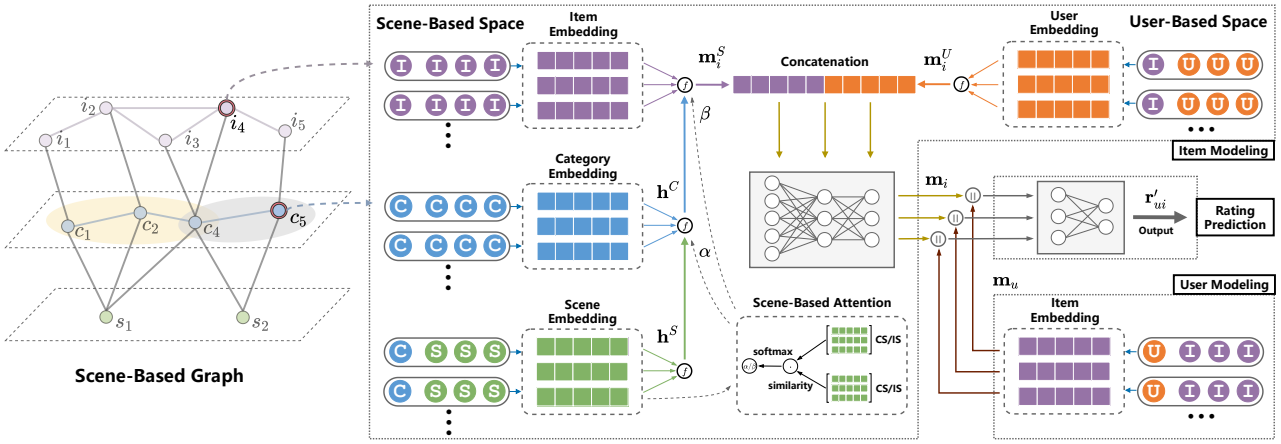


Figure 2: The illustration of SceneRec architecture (the arrowed lines present the bottom-up information flow). The embeddings of users and items are learned by user modeling and item modeling, respectively.

where $UI(u_p)$ denotes the set of items that are connected to user u_p, $\mathbf{e}_{i_q}$ is the embedding vector of item i_q, and σ is the nonlinear activation function. $\mathbf{W_u}$ and $\mathbf{b_u}$ are the weight matrix and the bias vector to be learned.

4.3 Item Modeling

The general representation $\mathbf{m}_{i_p}$ for item i_p can be further split into two parts: the embedding $\mathbf{m}^U_{i_p}$ in the user-based space and the embedding $\mathbf{m}^S_{i_p}$ in the scene-based space.

4.3.1 User-based embedding. In the user-item graph, an item i_p has connections with a set of users. We learn its embedding $\mathbf{m}^U_{i_p}$ by aggregating the embedding of these engaged users:

$$\mathbf{m}^U_{i_p} = \sigma(\mathbf{W_{iu}} \cdot \left\{ \sum_{u_q \in IU(i_p)} \mathbf{e}_{u_q} \right\} + \mathbf{b_{iu}}), \tag{2}$$

where $IU(i_p)$ denotes the set of users that are connected to item i_p, $\mathbf{e}_{u_q}$ is the embedding vector of user u_q, $\mathbf{W_{iu}}$ and $\mathbf{b_{iu}}$ are parameters to be learned. Since $\mathbf{m}^U_{i_p}$ is aggregated from user neighbors, $\mathbf{m}^U_{i_p}$ represents the user-based embedding of item i_p.

4.3.2 Scene-based embedding. In the scene-based graph, each item is connected to both other items and its category. So, the scene-based embedding $\mathbf{m}^S_{i_p}$ for item i_p is composed of representation that is specific to item neighbors and category neighbors.

For the category-specific representation, we should first generate the latent factor of each category. Since one category node can connect to both scene nodes and other related category nodes, the category representation can be further split into two types: the scene-specific and category-specific representation.

Given a category c_p, it may belong to a set of scenes and its scene-specific embedding vector $\mathbf{h}^S_{c_p}$ can be updated as follows:

$$\mathbf{h}^S_{c_p} = \sum_{s_q \in CS(c_p)} \mathbf{e}_{s_q}, \tag{3}$$

where $CS(c_p)$ is the set of scenes that category c_p belongs to and $\mathbf{e}_{s_q}$ is the embedding vector of scene s_q.

Besides the connection between scene nodes and category nodes, our model also captures the interactions between different category nodes. Each category contributes to the category-specific representation but categories do not always affect each other equally. Therefore, we apply the attention mechanism to learn the influence between different item categories. In this way, the category-specific representation $\mathbf{h}^C_{c_p}$ of the category c_p can be aggregated as follows:

$$\mathbf{h}^C_{c_p} = \sum_{c_q \in CC(c_p)} \alpha_{pq} \mathbf{e}_{c_q}, \tag{4}$$

where $CC(c_p)$ is the set of neighbor categories, $\mathbf{e}_{c_q}$ is the embedding vector of c_q, and α_{pq} is the attention weight. For a pair of categories, the more scenes they share, the higher relevance between them. Therefore, we propose a scene-based attention function to compute α_{pq}. Specifically, we calculate the attention score by comparing the sets of scenes that c_p and c_q belong to:

$$\alpha^*_{pq} = f\left(\sum_{s_a \in CS(c_p)} \mathbf{e}_{s_a}, \sum_{s_b \in CS(c_q)} \mathbf{e}_{s_b} \right), \tag{5}$$

where $f(\cdot)$ is an attention function to measure the input similarity. For simplicity, we use cosine similarity as $f(\cdot)$ in this work. α_{pq} is obtained by further normalizing α^*_{pq} via the softmax function:

$$\alpha_{pq} = \frac{\exp\left(\alpha^*_{pq}\right)}{\sum_{\{q|\forall c_q \in CC(c_p)\}} \exp\left(\alpha^*_{pq}\right)}. \tag{6}$$

Finally, we generate the overall representation $\mathbf{m}_{c_p}$ of category c_p by integrating the scene-specific representation and the category-specific representation:

$$\mathbf{m}_{c_p} = \sigma\left(\mathbf{W_{ic}} \cdot [\mathbf{h}^s_{c_p} \| \mathbf{h}^c_{c_p}] + \mathbf{b_{ic}}\right), \tag{7}$$

where $\|$ denotes the concatenation operation, $\mathbf{W_{ic}}$ and $\mathbf{b_{ic}}$ are parameters to be learned.

For item i_p, it is only connected to one pre-defined category and thus its category-specific representation $\mathbf{h}^C_{i_p}$ is denoted as:

$$\mathbf{h}^C_{i_p} = \mathbf{m}_{C(i_p)}, \tag{8}$$

where $C(i_p)$ indicates the category of i_p.

We continue to learn the item-specific representation $\mathbf{h}^I_{i_p}$ since there exist connections between different item nodes. Similar to category-category relations, items do not always affect each other equally and we apply the attention mechanism to learn $\mathbf{h}^I_{i_p}$:

$$\mathbf{h}^I_{i_p} = \sum_{i_q \in II(i_p)} \beta_{pq} \mathbf{e}_{i_q}, \tag{9}$$

Table 1: Statistics of JD datasets. Each relation A-B has three parts: number of A, number of B, and number of A-B.

Relations (A-B)	Baby & Toy	Electronics	Fashion	Food & Drink
User-Item	4,521-51,759 (481,831)	3,842-52,025 (539,066)	3,959-53,005 (541,238)	3,236-47,402 (463,391)
Item-Item	51,759-51,759 (3,002,806)	52,025-52,025 (2,992,333)	53,005-53,005 (2,750,495)	47,402-47,402 (2,606,003)
Item-Category	51,759-103 (51,759)	52,025-78 (52,025)	53,005-91 (53,005)	47,402-105 (47,402)
Category-Category	103-103 (1,791)	78-78 (825)	91-91 (1,058)	105-105 (1,628)
Scene-Category	323-103 (1,370)	54-78 (281)	438-91 (1,646)	136-105 (630)

where β_{pq} denotes the attention weight. Since items that belong to the same category share similarity, we leverage scene information to calculate β_{pq} by comparing their categories via the scene-based attention mechanism:

$$\beta^*_{pq} = f\left(\sum_{s_a \in IS(i_p)} \mathbf{e}_{s_a}, \sum_{s_b \in IS(i_q)} \mathbf{e}_{s_b}\right), \tag{10}$$

$$\beta_{pq} = \frac{\exp\left(\beta^*_{pq}\right)}{\sum_{\{q|\forall i_q \in II(i_p)\}} \exp\left(\beta^*_{pq}\right)}, \tag{11}$$

where $IS(i_p)$ is the set of scenes that contain item i_p's category.

In the end, we concatenate the category-specific representation $\mathbf{h}^C_{i_p}$ and the item-specific representation $\mathbf{h}^I_{i_p}$ to derive the overall representation $\mathbf{m}^S_{i_p}$ of the item i_p in the scene-based space:

$$\mathbf{m}^S_{i_p} = \sigma\left(\mathbf{W_{ii}} \cdot [\mathbf{h}^C_{i_p} \| \mathbf{h}^I_{i_p}] + \mathbf{b_{ii}}\right), \tag{12}$$

where $\mathbf{W_{ii}}$ and $\mathbf{b_{ii}}$ are parameters to be learned.

4.3.3 The general item embedding. The item embedding $\mathbf{m}^U_{i_p}$ in the user-based space learns the collaborative signals from user-item interactions, while the item embedding $\mathbf{m}^S_{i_p}$ in the scene-based space provides additional information from the scene-based graph. These two types of representations could be complementary to each other, and they are combined by a multilayer perceptron (MLP) to generate the general item embedding as follows:

$$\mathbf{m}_{i_p} = \mathbf{F}\left(\mathbf{W_i} \cdot [\mathbf{m}^U_{i_p} \| \mathbf{m}^S_{i_p}] + \mathbf{b_i}\right), \tag{13}$$

where $\mathbf{F}(\cdot)$ is a MLP network, $\mathbf{W_i}$ and $\mathbf{b_i}$ are parameters.

4.4 Model Optimization

Given the representation of user u_p and the general representation of item i_q, the user preference is obtained via a MLP network:

$$\mathbf{r}'_{pq} = \mathbf{F}\left(\mathbf{W_r} \cdot [\mathbf{m}_{u_p} \| \mathbf{m}_{i_q}] + \mathbf{b_r}\right), \tag{14}$$

where $\mathbf{W_r}$ and $\mathbf{b_r}$ are parameters to be learned.

To optimize the model parameters, we apply the pairwise BPR loss [14], which takes into account the relative order between observed and unobserved user-item interactions and assigns higher prediction scores to observed ones. The loss function is as follow:

$$\Omega(\Theta) = \sum_{(p,x,y) \in O} -\ln \sigma\left(\mathbf{r}'_{px} - \mathbf{r}'_{py}\right) + \lambda \|\Theta\|_2^2, \tag{15}$$

where $O = \left\{(p,x,y)|(p,x) \in \mathcal{R}^+, (p,y) \in \mathcal{R}^-\right\}$ denotes the pairwise training data, $\mathcal{R}^+$ and $\mathcal{R}^-$ are the observed and unobserved user-item interactions, respectively. Θ denotes all trainable model parameters and λ controls ℓ_2 regularization to prevent overfitting.

To sum up, we have different entity types, i.e., user, item, category and scene, in the user-item bipartite graph and the scene-based graph. In the learning process, the user representation is learnt from interactions between users and items. The item latent factor is generated from two components: the representation in the user-based space and the representation in the scene-based space. Then the user embedding and the item embedding are integrated to make prediction via pairwise learning.

5 EXPERIMENTS

In this section, we evaluate SceneRec on 4 real-world E-commerce datasets and focus on the following research questions:

RQ1: How does SceneRec perform compared with state-of-the-art recommendation methods?

RQ2: How do different key components of SceneRec affect the model performance?

RQ3: How does the scene information benefit recommendation?

5.1 Datasets

To the best of our knowledge, there are no public datasets that describe scene-based graph for recommender systems. To evaluate the effectiveness of SceneRec, we construct 4 datasets, namely, Baby & Toy, Electronics, Fashion, and Food & Drink, from JD.com, one of the largest B2C E-commerce platform in China. In each dataset, we build the user-item bipartite graph and the scene-based graph from online logs and commodity information. Statistics of the above datasets are shown in Table 1 and more details are discussed next. We have released the codes and datasets (available at https://github.com/e09b47e1/Scene-Based_Recommendation).

We first build the user-item bipartite graph that by randomly sampling a set of users and items from online logs. A user is then connected to an item if she or he clicked the item.

Next we build the scene-based graph where three different nodes, i.e., item, category and scene, are taken as input. We first consider connections between different item nodes. In E-commerce systems, users perform various behaviors such as "view" and "purchase", which can be further used to construct item-item relations. In this work, we choose "view" to build the item-item connections. A view session is a sequence of items that are viewed by a user within a period of time and it is intuitive that two items should be highly relevant if they are frequently co-viewed. In the item layer, two items are linked if they are co-viewed by a user within the same session where the weight is the sum of co-occurrence frequency within 2 months. For each item, we rank all the connected items by the edge weight and at most top 300 connections are preserved. All time period and numbers of connection are empirically set based on the trade-off between the size of datasets and co-view relevance between items.

We then connect each item to its pre-defined category to build the item-category relations. We also consider connections between different category nodes as shown in the second layer of the scene-based graph. For example, in E-commerce systems, the category "Mobile Phone" is strongly related to the category "Phone Case" but has little relevance to the category "Washing Machine", and thus the first two categories are linked. To achieve this, we compute the co-view frequency within six months between each pair of category node, and only top 100 connections of each category is preserved. In the end, each pair is further labeled as 0 or 1 from consensus decision-making by three data labeling engineers to indicate if there exists relevance or not.

The last step of building the scene-based graph is to link category nodes to scene nodes. Each scene consists of a set of selected categories which can be manually coded by human experts (scene mining is our future work). Specifically, this procedure consists

Table 2: Comparisons with baselines and model variants.

	Baby & Toy		Electronics		Fashion		Food & Drink	
	NDCG@10	HR@10	NDCG@10	HR@10	NDCG@10	HR@10	NDCG@10	HR@10
BPR-MF	0.3117	0.5213	0.4005	0.6082	0.3142	0.5294	0.3663	0.5445
NCF	0.2232	0.3800	0.3324	0.5364	0.1518	0.3090	0.3068	0.4628
CMN	0.2136	0.3840	0.4447	0.6725	0.2616	0.4516	0.4028	0.5854
PinSAGE	0.2124	0.4145	0.2954	0.5200	0.1770	0.3724	0.2791	0.4798
NGCF	0.3679	0.6000	0.4308	0.6559	0.3361	0.5749	0.3487	0.5228
KGAT	0.3055	0.5421	0.3616	0.6172	0.3115	0.5580	0.3221	0.5093
SceneRec-noitem	0.3977	0.6475	0.4748	0.7007	0.3936	0.6454	0.4080	0.6029
SceneRec-nosce	0.4193	0.6617	0.4715	0.7156	0.3933	0.6499	0.4156	0.6074
SceneRec-noatt	0.3950	0.6357	0.4665	0.7053	0.3953	0.6410	0.4138	0.6154
SceneRec	**0.4298**	**0.6771**	**0.4926**	**0.7524**	**0.4220**	**0.6763**	**0.4266**	**0.6211**

of two steps. First, an expert team (about 10 operations staff) edits a set of scene candidates based on the corresponding domain knowledge. Then, a data labeling team which consists of 3 engineers refines the generated scenes based on the criteria that whether each scene is reasonable to reflect a real-life situation.

To sum up, there is a user-item bipartite graph and a scene-based graph in the constructed E-commerce datasets where we have different types of nodes, i.e., user, item, category and scene. The scene-based graph presents a 3-layer hierarchical structure. There exist multiple relations among items, categories and scenes that are derived from user behavior data, commodity information and manual labeling. Thus, the datasets have all the characteristics of networks we want to study as described in Section 3.

5.2 Baselines

SceneRec leverages scene information to learn the representation vector of users and items in recommendation. Therefore, we compare SceneRec against various recommendation methods or network representation learning methods.

(1) **BPR-MF** [14] is a benchmark matrix factorization (MF) model which takes the user-item graph as input and BPR loss is adopted.

(2) **NCF** [7] leverages multi-layer perceptron to learn non-linearities between user and item interactions in the traditional MF model.

(3) **CMN** [3] is a state-of-the-art memory-based model to capture both global and local neighborhood structure of latent factors.

(4) **PinSAGE** [23] learns node representations on the large-scale item-item network where the representation of one item can be aggregated by the representation of its neighbor nodes. Here, we directly apply PinSAGE on the input user-item bipartite graph.

(5) **NGCF** [20]: This is a state-of-the-art GNN-based recommendation method, which learns the high-order connectivities based on the network structure.

(6) **KGAT** [19] investigates the utility of KG into GNN-based collaborative filtering where each item is mapped to an entity in KG. In our experiments, we regard each scene as a special type of KG entity and link it to item nodes via the category node connection. In such cases, the scene-based graph is degraded to the one that contains only item-scene connections. The graph contains two types of relations: an item *belongs to* a scene and a scene *includes* an item.

(7) **SceneRec-noitem** is a variant of SceneRec by removing item-item interactions in the scene-based graph.

(8) **SceneRec-nosce** is a variant of SceneRec by removing both category and scene nodes, and thus the scene-based graph only includes relations between items.

(9) **SceneRec-noatt** is another variant of SceneRec by removing the attention mechanism between item-item relations and category-category relations.

5.3 Experimental Settings

We evaluate the model performance using the leave-one-out strategy as in [1, 7]. For each user, we randomly hold out one positive item that the user has clicked and sample 100 unobserved items to build the validation set. Similarly, we randomly choose another positive item along with 100 negative samples to build the test set. The remaining positive items form the training set.

In our experiments, we choose *Hit Ratio* (HR) and *Normalized Discounted Cumulative Gain* [15] (NDCG) as evaluation metrics. HR measures whether positive items are ranked in the top K scores while NDCG focuses more on hit positions by assigning higher scores to top results. For both metrics, a larger value indicates a better performance. We report the average performance over all users with $K = 10$.

The hyper-parameters of SceneRec are fine-tuned using the validation set. We apply RMSProp [4] as the optimizer where the learning rate is determined by a grid search among $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$ and the ℓ_2 normalization coefficient λ is determined by a grid search among $\{0, 10^{-6}, 10^{-4}, 10^{-2}\}$. For fair comparisons, the embedding dimension d is set to 64 for all methods except NCF. For NCF, d is set to 8 due to the poor performance in higher dimensional space. For NGCF and KGAT, the depth L is set to 4 since it shows competitive performance via the high-order connectivity.

5.4 Experimental Results

5.4.1 Performance Comparison (RQ1). Table 2 reports comparative results of SceneRec against all 6 baseline methods, and we have the following observations:

(1) In general, NGCF achieves better results than baseline methods that take the user-item bipartite graph as input. There are two main reasons. First, GNN can effectively capture the non-linearity relations from user-item collaborative behaviors via information propagation on the graph. Second, NGCF learns the high-order connectivities between different types of nodes as shown in [20].

(2) KGAT further adds KG information into recommender systems, but it does not obtain the best result. Note that the KG quality is essential to the model performance. In our work, there are no available KG attributes that match our datasets, so there is no additional information to describe network items. Furthermore, the simple item-scene connection loses rich relations, e.g. category-category interactions and item-item interactions, in the scene-based graph, and may not advance model prediction.

(3) The proposed framework SceneRec obtains best overall performance using different evaluation metrics. Specifically, SceneRec boosts (16.8%, 10.8%, 25.6%, 5.9%) for NDCG@10, and (12.9%, 11.9%, 17.6%, 6.1%) for HR@10 on datasets (Baby & Toy, Electronics, Fashion, and Food & Drink), compared with the best baseline. There are several main reasons. First, SceneRec considers multiple types of entity nodes. To be specific, SceneRec generates

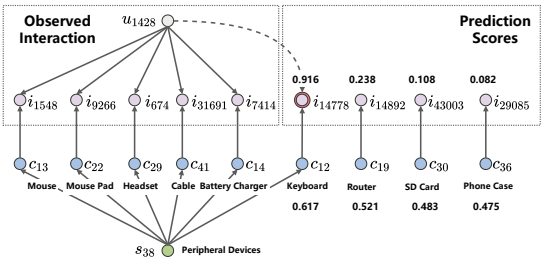


Figure 3: A real example on the Electronics dataset.

embedding representations of users and items from the user-item bipartite graph while it learns complementary representations of items from the scene-based graph, which is not accessible in baseline methods. Second, SceneRec creatively designs a principled hierarchical structure in the scene-based graph where additional scene-guided information is propagated into collaborative filtering. Third, SceneRec leverages GNN which captures local network structure to learn non-linear transformation of different types of graph nodes. Fourth, SceneRec adopts attention mechanism to attentively learn weighting importance among item-item connections and category-category connections.

5.4.2 Key Component Analysis (RQ2). Table 2 also reports comparative results against 3 variants and it is observed that:
(1) SceneRec-noitem obtains better experimental results than other baseline methods, and this indicates that the hierarchical structure of the scene-based graph can effectively propagate information and generate complementary scene-based representations. Moreover, SceneRec outperforms SceneRec-noitem and this verifies the effectiveness of incorporating item-item subnetwork into the scene-based graph.
(2) SceneRec-nosce outperforms all baselines because the item-item connections provide additional knowledge into conventional collaborative filtering. Comparing to SceneRec-nosce, SceneRec achieves better performance on both datasets and this indicates that, by leveraging scene information, SceneRec is capable of learning complementary representations beyond CF interactions.
(3) The prediction result of SceneRec is consistently better than that of SceneRec-noatt, and this verifies that the attention mechanism does benefit the recommendation by learning weights of 1-hop neighbors for each item node or each category node.

5.4.3 Case Study (RQ3). Finally, we use a case study to show the effects of integrating scene-specific representations into collaborative filtering in Figure 3. From the Electronics dataset, we randomly select a user u_{1428}, a set of items that the user has interacted with and a set of candidate items (whose prediction scores are given above item nodes). It is noted that we especially compute the average attention score (below the category node) between the candidate item and each item that the user has interacted with by the scene-based attentive mechanism.

The higher average attention score means more shared scenes between the candidate item and the user's interacted items. Therefore, the candidate item is more likely to occur in the scene derived from user interests, which could boost recommendation prediction. From this case study, we see that the average attention score does relate to the prediction result. For example, the positive sample of item i_{14778} that the user has interacted with has the highest prediction score and the largest average attention weight. Similar results can be also observed from other users. The item i_{14778} is recommended because its category "Keyboard" complements the user-interacted items' categories in the same scene "Peripheral Devices".

6 CONCLUSIONS

In this paper, we investigate the utility of integrating the scene information into recommender systems using graph neural networks, where a scene is formally defined as a set of pre-defined item categories. To integrate the scene information into graph neural networks, we design a principled 3-layer hierarchical structure to construct the scene-based graph and propose a novel method SceneRec. SceneRec learns item representation from the scene-based graph, which is further combined with the conventional latent representation learned from user-item interactions to make predictions. We conduct extensive experiments on four datasets that are collected from a real-world E-commerce platform. The comparative results and a case study demonstrate the rationality and effectiveness of SceneRec.

ACKNOWLEDGMENTS

This work is supported in part by National Key R&D Program of China 2018AAA0102301 and NSFC 61925203.

REFERENCES

[1] Jingyuan Chen, Hanwang Zhang, Xiangnan He, Liqiang Nie, Wei Liu, and Tat-Seng Chua. 2017. Attentive Collaborative Filtering: Multimedia Recommendation with Item- and Component-Level Attention. In *SIGIR*.
[2] Ernesto Diaz-Aviles, Mihai Georgescu, and Wolfgang Nejdl. 2012. Swarming to rank for recommender systems. In *RecSys*.
[3] Travis Ebesu, Bin Shen, and Yi Fang. 2018. Collaborative Memory Network for Recommendation Systems. In *SIGIR*.
[4] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. MIT Press.
[5] Marco Gori and Augusto Pucci. 2007. ItemRank: A Random-Walk Based Scoring Algorithm for Recommender Engines. In *IJCAI*.
[6] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NIPS*.
[7] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. In *WWW*.
[8] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative Filtering for Implicit Feedback Datasets. In *ICDM*.
[9] Wang-Cheng Kang, Eric Kim, Jure Leskovec, Charles Rosenberg, and Julian J. McAuley. 2019. Complete the Look: Scene-Based Complementary Product Recommendation. In *CVPR*.
[10] M. Hadi Kiapour, Kota Yamaguchi, Alexander C. Berg, and Tamara L. Berg. 2014. Hipster Wars: Discovering Elements of Fashion Styles. In *ECCV*.
[11] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.
[12] Ayangleima Laishram, Satya Prakash Sahu, Vineet Padmanabhan, and Siba Kumar Udgata. 2016. Collaborative Filtering, Matrix Factorization and Population Based Search: The Nexus Unveiled. In *ICONIP*.
[13] Ziwei Liu, Ping Luo, Shi Qiu, Xiaogang Wang, and Xiaoou Tang. 2016. DeepFashion: Powering Robust Clothes Recognition and Retrieval with Rich Annotations. In *CVPR*.
[14] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian Personalized Ranking from Implicit Feedback. In *UAI*.
[15] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor (Eds.). 2011. *Recommender Systems Handbook*. Springer.
[16] Rianne van den Berg, Thomas N. Kipf, and Max Welling. 2017. Graph Convolutional Matrix Completion. *CoRR* (2017).
[17] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. 2015. Collaborative Deep Learning for Recommender Systems. In *SIGKDD*.
[18] Hongwei Wang, Miao Zhao, Xing Xie, Wenjie Li, and Minyi Guo. 2019. Knowledge Graph Convolutional Networks for Recommender Systems. In *WWW*.
[19] Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. 2019. KGAT: Knowledge Graph Attention Network for Recommendation. In *KDD*.
[20] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. 2019. Neural Graph Collaborative Filtering. In *SIGIR*.
[21] Yinqing Xu, Wai Lam, and Tianyi Lin. 2014. Collaborative Filtering Incorporating Review Text and Co-clusters of Hidden User Communities and Item Groups. In *CIKM*.
[22] Xiwang Yang, Yang Guo, Yong Liu, and Harald Steck. 2014. A survey of collaborative filtering based social recommender systems. *Computer Communications* (2014).
[23] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *KDD*.
[24] Hui Zhang, Xu Chen, and Shuai Ma. 2019. Dynamic News Recommendation with Hierarchical Attention Network. In *ICDM*.

Short Paper

Efficient Contact Similarity Query over Uncertain Trajectories

Xichen Zhang, Suprio Ray, Farzaneh Shoeleh, Rongxing Lu
Canadian Institute for Cybersecurity, Faculty of Computer Science, University of New Brunswick
Fredericton, NB, Canada
{xichen.zhang,sray,farzaneh.shoeleh,rlu1}@unb.ca

ABSTRACT

The problem of studying the effective contact between different moving objects (MOs) has received great interest in recent years. However, the uncertain nature of trajectory data poses significant challenges for the researchers. Most of the existing studies focus on range query or similarity join. But they cannot find MOs who may contact each other for a long enough time in the same location. In this paper, we study how to evaluate the effective contact among MOs. More specifically, we provide a purposeful definition for measuring the contact effectiveness, called **Contact Similarity**. Based on this notion, we introduce a novel query called **Contact Similarity Query** (*CSQ*). A necklace-based model is used for representing uncertain trajectories, in which a bead (or an ellipse) indicates the possible locations of an object in-between two successive trajectory observations. To enable efficient query processing, we design a novel data structure called UTM-tree (i.e., Uncertain Trajectory M-tree) for indexing the necklace-based trajectories. Experiments have been conducted on a real-world dataset, and the results demonstrate that our proposed solution significantly outperforms the baseline approaches.

1 INTRODUCTION

With the increasing availability and rapid development of global positioning technologies, moving objects (MOs)' trajectories can be utilized by Location-Based Services (LBS) in many applications, such as vehicle navigation, traffic management, and co-occurrence analysis. However, due to sensor devices' physical and resource limitations or privacy considerations, MOs' trajectories are often captured at low sampling rates, and the time interval between two consecutive observations is quite long. In such uncertain courses, no information can be found about the whereabouts of MOs in-between two successive points. Recently, the problem of querying uncertain trajectory has been studied by many works, e.g., [1, 2, 5, 6, 10, 13]. However, most of them focused on retrieving qualified results regarding either probabilistic range query or similarity join. The existing approaches are not applicable to the problem of modeling the effective contact among individuals, which is a research topic of great importance. As follows, a real-world example is presented to describe a motivational scenario.

Example 1. *Fig. 1 shows the uncertain trajectories of three visitors m_1, m_2, and m_3 in one day. They are equipped with a GPS device in their car, and they periodically reported their real-time locations along their trajectories. But we don't know where they are in-between two consecutive reports. We can see that m_1 and m_3 may have a very high trajectory similarity since the spatial and temporal distance between their observations can be very small. However, m_1 and m_3 may never come in contact with each other. On the other hand, assume that along the trip, m_1 parked the car at location A, disembarked the car, walked into the Park and stayed there for a while. Then m_1 went back to A to board the car and continued with the trip. Perhaps m_2 can take similar action at location B, i.e., m_2 may also stay in the Park for a while during his/her trip. Therefore, even though m_1 and m_2 have very low trajectory similarity, they may have an effective contact either in the Park or in the Shopping District. But such a result can never be captured by the traditional trajectory similarity query. Besides, assume that several days later, m_1 found him/herself as an infected individual of a highly contagious disease (e.g., Covid 19). To help determine other susceptible persons who might be infected, m_1 provided more details about his/her trajectory to an authorized third-party, such as a local public health agency (LPHA). For example, m_1 stayed in the Park from 10:00 am to 10:50 am, and then went to the Shopping District from 11:35 am to 12:30 pm. Next, given this information, LPHA discovered that m_1 and m_2 may have possible contact with each other, and may like to know the effectiveness of their contact (e.g., the chance of their contact and how long they may contact with each other). Moreover, hundreds of people may have visited the neighboring areas on the same day. LPHA wants to find the susceptible individuals from the big group of candidates who may have effective contact with m_1.*

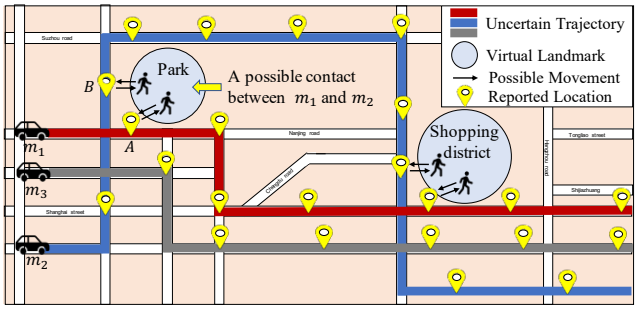


Figure 1: An real-world example of possible contacts between MOs m_1, m_2, and m_3

To address the problem described above, it is important to answer the following questions successively:

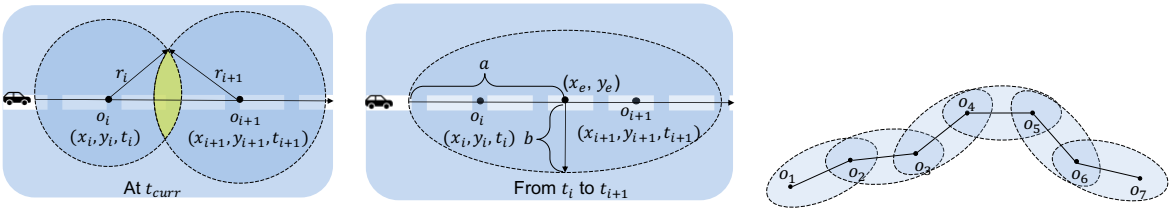
Q1: How to calculate the potential contact area and the possible longest contact time duration between m_1 and m_2?

Q2: How to measure the effective contact between m_1 and m_2?

Q3: How to efficiently retrieve the qualified results from a large number of people who visited the park on the same day as m_1?

To the best of our knowledge, little or no systematic and theoretical study has been conducted to address the abovementioned questions. Aiming to fill this research gap, we propose a study on analyzing the effective contact between MOs over uncertain trajectories. Specifically, the main contributions of this work are three-fold as follows.

• We propose a definition of **Contact Similarity** over uncertain trajectories and a novel query called **Contact Similarity Query** (*CSQ*). With *CSQ*, we can assess the effectiveness of contact between MOs in free space.



(a) The possible locations of the MO at time $t_{curr}(t_i \le t_{curr} \le t_{i+1})$

(b) All the possible locations (the bead B_i) of the MO between t_i and t_{i+1}.

(c) An example of necklace for a given uncertain trajectory with 7 observations.

Figure 2: An example of bead B_i given the information of o_i, o_{i+1}, and v_{max}.

Table 1: The summary of notations

Notation	Definition
Trajectory Notations	
MO	Moving object
$\mathsf{T} = \{o_1, o_2, ..., o_n\}$	An uncertain trajectory
$t_i/(x_i, y_i)$	Timestamp/coordinates of o_i in T
v_{max}	The maximum speed of a MO
T_{neck}	The necklace of T
B_i	A bead in T_{neck}
P	A virtual landmark
t^s/t^e	The arrival/departure time of P
Query Notations	
R_{int}	The intersection area ratio
$\tau(\mathsf{P}, \mathsf{B}_i)$	The possible longest time duration
$Sim_{cont}^{\mathsf{P}\cap\mathsf{B}_i}$	Contact similarity at bead level
D	A MO database
$Sim_{cont}^{\mathsf{P}\cap\mathsf{T}}$	Contact similarity at trajectory level
$\bar{t}^s$	The earliest arrival time
$\bar{t}^e$	The latest departure time

• We design a novel data structure called UTM-tree (Uncertain Trajectory M-tree) for indexing uncertain trajectories, which significantly improves the efficiency for processing *CSQ*.

• We conduct an extensive experimental evaluation of the proposed approaches, demonstrating the effectiveness and efficiency of the methodology proposed in this work.

The rest of this paper is organized as follows. In Section 2, we introduce the preliminaries and problem definition. In Section 3, we propose the novel indexing structure for managing uncertain trajectories. In Section 4, we present the details of our experiments and the evaluation results, followed by the related works in Section 5. Finally, we recap the conclusions in Section 6.

2 PRELIMINARIES

In this section, we briefly introduce the preliminaries of this work. The frequently used notations are listed in Table 1.

2.1 Trajectories, Beads, and Necklaces

We first introduce the definitions of trajectory, beads, and necklaces. A trajectory is defined as follows.

Definition 1. *A **trajectory** T of a MO is defined as finite, time-ordered observations $\mathsf{T} = \{o_1, o_2, ..., o_n\}$, where $o_i = (x_i, y_i, t_i)$ for $i \in [1, n]$, with (x_i, y_i) being a sample point in Euclidean space, and t_i being a timestamp.*

The possible locations of a MO in-between two observations can be defined by a bead (or ellipse) [10] as follows.

Definition 2. *Let v_{max} denote the maximum speed that an object can take between o_i and o_{i+1}. A **bead** $\mathsf{B}_i = \{(x_i, y_i, t_i), (x_{i+1}, y_{i+1}, t_{i+1})\}$ is defined as all points (x, y, t) that satisfy the following constraints:*

$$\begin{cases} t_i \le t \le t_{i+1} \\ (x - x_i)^2 + (y - y_i)^2 \le (t - t_i)^2 \times v_{max}^2 \\ (x - x_{i+1})^2 + (y - y_{i+1})^2 \le (t_{i+1} - t)^2 \times v_{max}^2. \end{cases} \tag{1}$$

From Fig. 2 (a), we can see that from time t_i to t_{curr} ($t_{curr} > t_i$), the MO's possible travel area is a circle with (x_i, y_i) being the center and $r_i = (t_{curr} - t_i) \times v_{max}$ being the radius. Similarly, from time t_{curr} to t_{i+1} ($t_{curr} < t_{i+1}$), the possible locations are included in a circle centered at (x_{i+1}, y_{i+1}) with radius $r_{i+1} = (t_{i+1} - t_{curr}) \times v_{max}$. So, the overlapped area in Fig. 2 (a) of the two circles includes MO's possible locations at current time t_{curr}. Based on [8], all the possible locations from time t_i to t_{i+1} form an ellipse (or bead) with foci at (x_i, y_i) and (x_{i+1}, y_{i+1}) (see in Fig. 2 (b)). According to [10], the equation of the bead B is:

$$(x - x_e)^2/a^2 + (y - y_e)^2/b^2 = 1, \tag{2}$$

where (x_e, y_e) is the center of the ellipse; a and b represents the semi-major axis and semi-minor axis of the ellipse, as follows.

$$x_e = \frac{x_i + x_{i+1}}{2},\ y_e = \frac{y_i + y_{i+1}}{2},\ a = \frac{v_{max} \times (t_{i+1} - t_i)}{2}$$
$$b = \frac{\sqrt{v_{max}^2 \times (t_{i+1} - t_i)^2 - (x_{i+1} - x_i)^2 - (y_{i+1} - y_i)^2}}{2}. \tag{3}$$

Definition 3. *A trajectory $\mathsf{T} = \{o_1, o_2, ..., o_n\}$ can be represented as a sequence of beads, which is called a **necklace** and denoted as $\mathsf{T}_n = \{\mathsf{B}_1, \mathsf{B}_2, ..., \mathsf{B}_{n-1}\}$ (see an example in Fig. 2 (c)), such that o_i and o_{i+1} form the bead B_i for $i \in [1, n-1]$.*

2.2 Contact Similarity

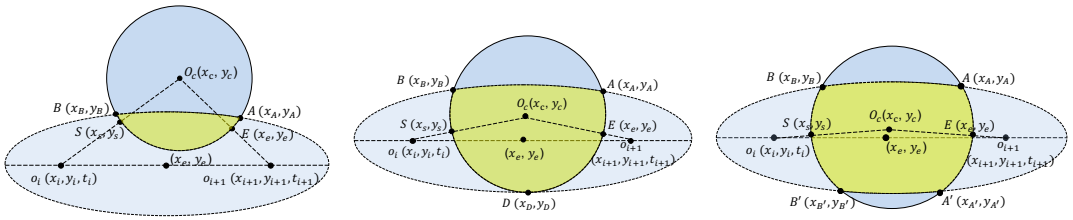
In this section, we introduce the essential definitions for contact similarity.

Definition 4. *A **virtual landmark** P is a place where two MOs can contact with each other. P can be either meaningful locations (e.g., a shopping district or a park) or non-meaningful locations (e.g., a crossroad). For simplicity sake, any P in this work is considered as a circle that is centered at $O_c = (x_c, y_c)$ with a given radius r_c. Hence, the equation of P is $(x - x_c)^2 + (y - y_c)^2 = r_c^2$.*

Definition 5. *For a MO m_1, he/she stayed at P for a certain period time. For another MO m_2, his/her necklace is $\mathsf{T}_n = \{\mathsf{B}_1, \mathsf{B}_2, ..., \mathsf{B}_{n-1}\}$. If P spatially overlaps with one bead B_i for $i \in [1, n-1]$ (denoted as $\mathsf{P} \cap_S \mathsf{B}_i \neq \emptyset$), then the **intersection area ratio** R_{int} is*

$$\mathsf{R}_{int} = \mathsf{Area}_{int}/\mathsf{Area}_P, \tag{4}$$

where Area_{int} is the intersection area between the P and B_i, and Area_P is the area of P.



(a) *Two* intersection points (A, B) (b) *Three* intersection points (A, B, D) (c) *Four* intersection points (A, B, A', B')

Figure 3: Examples of overlap between a virtual landmark and a bead with different intersection points

Fig. 3 shows different scenarios that P (the circle) overlaps with a bead B (the ellipse), with two, three, and four intersection points, respectively. Then, we difine the possible longest contact time duration between MOs as follows.

Definition 6. *For MO m_1, he/she stayed at* P *from time t^s to t^e. For another MO m_2, his/her necklace $\mathsf{T}_n = \{\mathsf{B}_1, \mathsf{B}_2, ..., \mathsf{B}_{n-1}\}$. Moreover, if one of the beads B_i (for $i \in [1, n-1]$) temporally overlaps with* P *(denoted as $\mathsf{P} \cap_T \mathsf{B}_i \neq \emptyset$), then the **possible longest time duration** $\tau(\mathsf{P}, \mathsf{B}_i)$ between m_1 and m_2 can be calculated as*

$$\tau(\mathsf{P}, \mathsf{B}_i) = \min\{t^e, \bar{t}^e\} - \max\{t^s, \bar{t}^s\}, \tag{5}$$

where $\bar{t}^s$ means the possible earliest time for m_2 to arrive at P*, and $\bar{t}^e$ means the possible latest time for m_2 to leave* P*.*

Example 2. *If m_1 stayed at* P *from t^s = 10:00 am and t^e = 10:50 am. For m_2, $\bar{t}^s$ = 10:05 am and $\bar{t}^e$ = 10:35 am, then $\tau(\mathsf{P}, \mathsf{B}_i)$ = min{10:50, 10:35} - max{10:00, 10:05} = 30 min.*

Next, inspired by [7], we formally propose the definition of contact similarity as below.

Definition 7. *For m_1, he/she stayed at* P *from time t^s to t^e. For m_2, his/her necklace is $\mathsf{T}_n = \{\mathsf{B}_1, \mathsf{B}_2, ..., \mathsf{B}_{n-1}\}$. If $\forall \mathsf{B}_i \in \mathsf{T}_n$ for $i \in [i, n-1]$ such that $\mathsf{P} \cap_S \mathsf{B}_i \neq \emptyset$ and $\mathsf{P} \cap_T \mathsf{B}_i \neq \emptyset$, then **the contact similarity at bead level** $Sim_{cont}^{\mathsf{P}\cap\mathsf{B}_i}$ between m_1 and m_2 at* P *in terms of B_i can be calculated as:*

$$Sim_{cont}^{\mathsf{P}\cap\mathsf{B}_i} = 1 - (1 - \mathsf{R}_{int})^{\tau(\mathsf{P}, \mathsf{B}_i)} \tag{6}$$

We assume that the intersection area ratio R_{int} indicates the possible opportunities that m_1 may be in contact with m_2 at P per unit time. Therefore, $(1 - \mathsf{R}_{int})^{\tau(\mathsf{P}, \mathsf{B}_i)}$ denotes the probability that m_1 and m_2 have no effective contact during the time period $\tau(\mathsf{P}, \mathsf{B}_i)$. Consequently, $Sim_{cont}^{\mathsf{P}\cap\mathsf{B}_i}(m_1, m_2) = 1 - (1 - \mathsf{R}_{int})^{\tau(\mathsf{P}, \mathsf{B}_i)}$. A larger value of $Sim_{cont}^{\mathsf{P}\cap\mathsf{B}_i}(m_1, m_2)$ indicates a higher probability that an effecitive contact may happen between m_1 and m_2 at P.

Definition 8. *Let $\mathbb{B}$ denote a set of beads in T_n, such that for $\forall \mathsf{B}_i \in \mathbb{B}$, $\mathsf{P} \cap_S \mathsf{B}_i \neq \emptyset$ and $\mathsf{P} \cap_T \mathsf{B}_i \neq \emptyset$. Then, **the contact similarity at trajectory level** $Sim_{cont}^{\mathsf{P}\cap\mathsf{T}}$ can be calculated as:*

$$Sim_{cont}^{\mathsf{P}\cap\mathsf{T}} = \frac{\sum_{\mathsf{B}_i \in \mathbb{B}} Sim_{cont}^{\mathsf{P}\cap\mathsf{B}_i}}{|\mathbb{B}|}. \tag{7}$$

Here, $|\mathbb{B}|$ denotes the number of beads in $\mathbb{B}$.

2.3 Problem Definition

Definition 9. *For m_1, the **query trajectory** T_q is defined as a sequence of* P*(s) that m_1 visited, i.e., $\mathsf{T}_q = \{\mathsf{P}_1, \mathsf{P}_2, ..., \mathsf{P}_d\}$. $\mathsf{P}_k = (x_k^c, y_k^c, r_k^c, t_k^s, t_k^e)$ for $k \in [1, d]$. Here, m_1 stayed at P_k from t_k^s to t_k^e, (x_k^c, y_k^c) and r_k^c are the center and radius of P_k, respectively.*

Definition 10. *Given a query trajectory $\mathsf{T}_q = \{\mathsf{P}_1, \mathsf{P}_2, ..., \mathsf{P}_d\}$, a MO database D that contains s necklaces (e.g., $D = \{\mathsf{T}_{n1}, \mathsf{T}_{n2}, ..., \mathsf{T}_{ns}\}$), and a predefined threshold α, for each $\mathsf{P}_k \in \mathsf{T}_q$ where $k \in [1, d]$, the **Contact Similarity Query** (CSQ) finds all the necklaces $\mathsf{T}_{nj} \in D$ for $j \in [1, s]$, such that $Sim_{cont}^{\mathsf{P}_k\cap\mathsf{T}_{nj}} \geq \alpha$.*

Example 3. *In Fig. 1, T_q is m_1's query trajectory. For example, m_1 stayed in the park P_1 from 10:00 to 10:50 am, then m_1 went to the shopping district P_2 and stayed there from 11:35 am to 12:30 pm. D is a MO database that contains 1000 people's trajectory necklaces who visited the neighboring areas on the same day, and let $\alpha = 0.8$. For every virtual landmark P_k that m_1 visited, the CSQ finds all the trajectory necklaces $\mathsf{T}_{nj} \in D$ such that $Sim_{cont}^{\mathsf{P}_k\cap\mathsf{T}_{nj}} \geq 0.8$.*

2.4 Calculation of $\mathtt{Area}_{int}$ and $\tau(\mathsf{P}, \mathsf{B}_i)$

In this section, we discuss how to calculate $\mathtt{Area}_{int}$ and $\tau(\mathsf{P}, \mathsf{B}_i)$, followed by the workflow for *CSQ*.

Given the P with equation $(x - x_c)^2 + (y - y_c)^2 = r_c^2$ and the B_i with equation $(x - x_e)^2/a^2 + (y - y_e)^2/b^2 = 1$, it is straightforward to calculate $\mathtt{Area}_{int}$ as follows.

• There are two or three intersection points between the virtual landmark P and the bead B_i (see Fig. 3 (a) and (b)).

$$\mathtt{Area}_{int} = \int_{x_B}^{x_A} \left(\frac{b}{a}\sqrt{a^2 - (x - x_e)^2} + y_e + \sqrt{r_c^2 - (x - x_c)^2} - y_c\right) dx \tag{8}$$

• There are four intersection points between the virtual landmark P and the bead B_i (see Fig. 3 (c)).

$$\begin{aligned}\mathtt{Area}_{int} = &\pi r_c^2 - \int_{x_B}^{x_A} \left(\sqrt{r_c^2 - (x - x_c^2)} + y_c - \frac{b}{a}\sqrt{a^2 - (x - x_e)^2} - y_e\right) dx \\ &- \int_{x_B'}^{x_A'} \left(y_e - \frac{b}{a}\sqrt{a^2 - (x - x_e)^2} + \sqrt{r_c^2 - (x - x_c)^2} - y_c\right) dx\end{aligned} \tag{9}$$

In Fig. 3, we can easily know that S is the earliest point that one MO can arrive at P, and E is the latest point that MO can leave P. Therefore, given the maximum speed v_{max}, we can calculate $\bar{t}^s$ and $\bar{t}^e$ as follows.

$$\bar{t}^s = t_i + \frac{|o_i S|}{v_{max}},\ \bar{t}^e = t_{i+1} - \frac{|o_{i+1} E|}{v_{max}} \tag{10}$$

Based on $\bar{t}^s$ and $\bar{t}^e$, we can calculate $\tau(\mathsf{P}, \mathsf{B}_i)$ by Eq. (5). After calculating R_{int} and $\tau(\mathsf{P}, \mathsf{B}_i)$, given an uncertain trajectory necklace database D and a trajectory query T_q, we can compute $Sim_{cont}^{\mathsf{P}\cap\mathsf{B}_i}$ and $Sim_{cont}^{\mathsf{P}\cap\mathsf{T}}$ by Eq. (6) and (7), respectively. If $Sim_{cont}^{\mathsf{P}\cap\mathsf{T}} \geq \alpha$, the result should be returned. We repeat the above mentioned process untill all the necklaces have been visited.

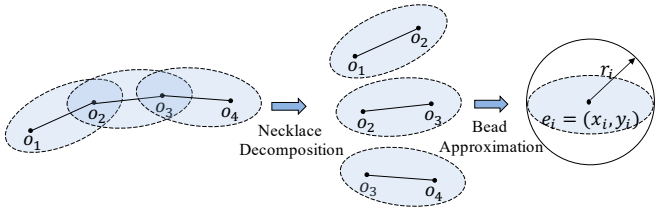


Figure 4: Decomposition of necklaces into circles

3 THE INDEXING STRUCTURE FOR *CSQ*

M-tree [4] is a popular indexing structure for efficient range query and k-nearest neighbor query. But we cannot directly adopt it in this work, because M-tree cannot handle entries like ellipse or beads. Therefore, based on M-tree, we propose a novel indexing structure called Uncertain Trajectory M-tree (UTM-tree), which is discussed next.

3.1 Decomposing the Necklace

First, we decompose the necklaces into beads. After decomposition, we use a circle to approximate each bead, and the circle is centered at the center of the bead (x, y) with radius $r = a/2$ (see in Fig. 4). Each circle is considered an individual entry and is stored independently in the leaf node of the UTM-tree.

Algorithm 1: The Inserion Algorithm: Insert(E_i, N_j)

Input : A new entry $E_i = [e_i, r_i, t_i, t_{i+1}, \texttt{TID}, \texttt{Ptr}(E_p), \texttt{Ptr}(E_n)]$, a tree node $N_j = [E_j, R_j, t_{min}, t_{max}, \texttt{ptr}]$
Output: A UTM-tree with E_i added

```
if N_j is not a leaf node then
    for all the child nodes N_j^sub of N_j do
        Let R_j^sub denote the covering radius of N_j^sub, calculate d(E_i, N_j^sub);
        if ∃ at least one N_j^sub s.t. d(E_i, N_j^sub) + r_i ≤ R_j^sub then
            Insert(E_i, N_j^sub) where d(E_i, N_j^sub) is minimun;
        else
            Insert(E_i, N_j^sub) where d(E_i, N_j^sub) + r_i − R_j^sub is minimum;
else
    if N_j is not full then
        Add E_i into N_j, update t_min and t_max;
        if d(E_i, N_j) + r_i > R_j then R_j = d(E_i, N_j) + r_i ;
    else Split(E_i, N_j) ;
return An updated UTM-tree after inserting E_i into N_j
```

3.2 Building the UTM-tree

Next, we describe how to construct a UTM-tree, including the format of an entry/node, entry insertion, splitting policy, selection of routing objects, and query processing. An illustration of the UTM-tree is shown in Fig. 5.

3.2.1 Format of the entry. Same as the M-tree, in a UTM-tree, all the entries are stored in the leaf nodes. And also, each node in the tree can store at most M entries, which is also called the capacity of the tree. Specifically, we consider each circle as the entry of the UTM-tree, and each entry is in format of E_i. Here, $E_i = [e_i, r_i, t_i, t_{i+1}, \texttt{TID}, \texttt{Ptr}(E_p), \texttt{Ptr}(E_n)]$, where $e_i = (x_i, y_i)$ and r_i are the center and radius of the approximating circle, t_i and t_{i+1} are the observation timestamps from the original trajectory that form the bead, TID is the trajectory identifier which the bead belongs to, and $\texttt{Ptr}(E_p)$ and $\texttt{Ptr}(E_n)$ are doubly linked list pointers for the previous/next entry that corresponding to beads in the original trajectory.

In the UTM-tree, each node selects an entry from the leaf node as its routing object (similar to M-tree). The format of a node is represented as $N_j = [E_j, R_j, t_{min}, t_{max}, \texttt{ptr}(N_j^{sub})]$, where E_j is its routing object, R_j is the covering radius that covers all the entries stored in N_j. t_{min} and t_{max} are the minimum and maximum time for all the entries stored in N_j, and ptr is a pointer pointing to its child node N_j^{sub} if N_j is a non-leaf node.

3.2.2 Insert an Entry. First, we define the distance between an entry E_i and a node N_j as $d(E_i, N_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$, where (x_i, y_i) is the center of E_i, and (x_j, y_j) is the center of the rounting object of N_j. The insert algorithm recursively descends the tree to locate the most suitable leaf node for E_i. We first find the leaf node such that after adding E_i, no enlargement of the covering radius is needed, i.e., $d(E_i, N_j) + r_i \le R_j$. If more than one child nodes are found, then the node that is closest to E_i will be chosen. However, if no such node exists, the choice is to minimize the increase of the covering radius, i.e., $d(E_i, N_j) + r_i - R_j$ is minimum. After inserion, if $d(E_i, N_j) + r_i > R_j$, then we update $R_j = d(E_i, N_j) + r_i$. Besides, t_{min} and t_{max} for all the visited nodes should be updated as well. After inserting all the entries into the UTM-tree, we use a doubly linked list to connect those from the same trajectory (see in Fig. 5). The details of the insertion algorithm is shown in Algorithm 1.

Algorithm 2: The Split Algorithm: Split(E_i, N_j)

Input : An entry $E_i = [e_i, r_i, t_i, t_{i+1}, \texttt{TID}, \texttt{Ptr}(E_p), \texttt{Ptr}(E_n)]$, a leaf node $N_j = [E_j, R_j, t_{min}, t_{max}, \texttt{ptr}]$
Output: A UTM-tree that splits N_j into N_j^1 and N_j^2

```
Let N_j be the set of all N_j's entries plus E_i;
Select two routing objects E_j^1 and E_j^2 from N_j based on m_RAD algorithm [4], and then partition N_j into two sets, N_j^1 and N_j^2;
Allocate a new node N_j', store N_j^1 in N_j and N_j^2 in N_j';
Replace N_j's rounting object E_j with E_j^1, update N_j's R_j;
Set E_j^2 as N_j''s rounting object, and update N_j''s R_j';
if N_j is not the root of the tree (N_j^par is N_j's parent) then
    if N_j^par is full then Split(N_j', N_j^par) ;
    else add N_j' into N_j^par;
else Allocate a new root node N_r, set N_r as parent node for N_j and N_j' ;
Update t_min and t_max for all the visited nodes;
return An updated UTM-tree after spliting N_j by E_i
```

3.2.3 Split a Node. E_i cannot be inserted into a leaf node N_j if there are M entries in N_j. we need to split N_j into two nodes. Particularly, let $\mathcal{N}_j$ denote the set of all N_j's entries plus E_i. We find two rounting objects E_j^1 and E_j^2 from $\mathcal{N}_j$ and partition N_j into two nodes N_j^1 and N_j^2 based on the *m_RAD* algorithm [4]. After partitioning, the sum of the covering radius $R_j^1 + R_j^2$ is miminum. The split algorithm is shown in Algorithm 2.

3.2.4 Data Filtering for CSQ Query. In this part, we discuss how to filter qualified candidates for *CSQ* using UTM-tree.

Given a root node N_r and a query trajectory $\mathsf{T}_q = \{\mathsf{P}_1, \mathsf{P}_2, ..., \mathsf{P}_d\}$ where $\mathsf{P}_k = (x_k^c, y_k^c, r_k^c, t_k^s, t_k^e)$ for $k \in [1, d]$, we can first filter all the entries $E_i = [e_i, r_i, t_i, t_{i+1}, \texttt{TID}, \texttt{Ptr}(E_p), \texttt{Ptr}(E_n)]$ such that (1) $d(e_i, (x_k^c, y_k^c)) \le r_i + r_k^c$ and (2) $t_i \le t_k^s$ and $t_{i+1} \ge t_k^c$ for all pairs of (N_r, P_k). Next, based on the filtered entries, we can get a list of trajectory identifiers TID (s). For each trajectory T from this list, we can calculate $Sim_{cont}^{\mathsf{P}_k \cap \mathsf{T}}$, and then check if $Sim_{cont}^{\mathsf{P}_k \cap \mathsf{T}} \ge \alpha$. The UTM-tree indexing structure can significantly reduce the

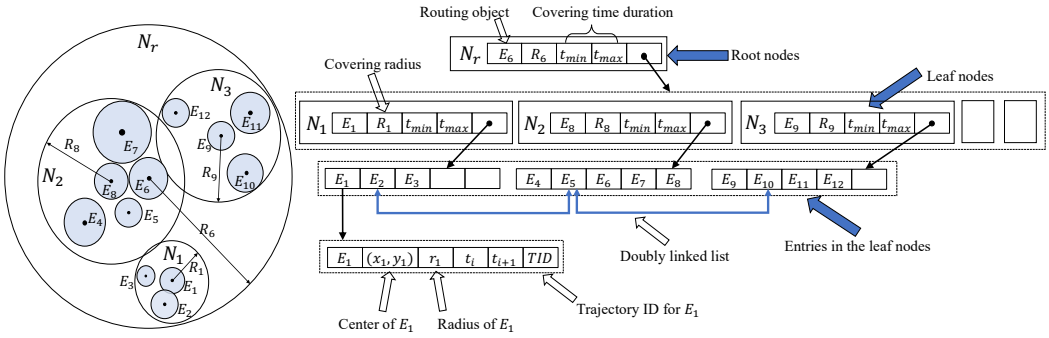


Figure 5: An example of the UTM-tree with capacity of 5

number of trajectories to be visited, thus it can greatly increase the performance of the *CSQ* query. The details of the data filtering algorithm is shown in Algorithm 3.

Algorithm 3: The filtering algorithm: Filter(N_r, P_k)

Input : A root node N_r, and P_k where $\mathsf{P}_k = (x_k^c, y_k^c, r_k^c, t_k^s, t_k^e)$
Output: A list of the trajectories IDs that satisfy the query

if N_r *is not a leaf node* **then**
  for *all the children nodes* N_r^{sub} *in* N_r *where* $N_r^{sub} = [E_r^{sub}, R_r^{sub}, t_{min}^{sub}, t_{max}^{sub}, \mathtt{ptr}(N_r^{sub(sub)})]$ **do**
    if $(t_{min}^{sub}, t_{max}^{sub}) \cap (t_k^s, t_k^e) \neq \emptyset$ **then**
      if $d(E_r^{sub}, (x_k^c, y_k^c)) \leq r_k^c + R_r^{sub}$ **then** Filter(N_r^{sub}, P_k);
      else Prune N_r^{sub};
    else Prune N_r^{sub};
else
  for *every* $E_i = [e_i, r_i, t_i, t_{i+1}, \mathtt{TID}, \mathtt{Ptr}(E_p), \mathtt{Ptr}(E_n)]$ *in* N_r **do**
    if $(t_i, t_{i+1}) \cap (t_k^s, t_k^e) \neq \emptyset$ **then**
      if $d(e_i, (x_k^c, y_k^c)) \leq r_k^c + r_i$ **then** Add TID into an ID list;
      else Prune E_i;
    else Prune E_i;
return *The ID list*

4 EXPERIMENTAL EVALUATION

We implemented the systems with Python on an Intel(R) Core(TM) i7-6700 CPU @3.60GHz running Windows 64-bit OS with 32 GB RAM. The details of experimental settings and the performance evaluation are discussed as follows.

4.1 Settings

In this part, we discuss the datasets, experimental settings, and the basic approaches in this work.

• Beijing Taxi Dataset: This dataset [12] contains the GPS trajectories of 10357 taxis from Feb. 2 to Feb. 8, 2008 in Beijing. The total number of points in this dataset is about 15 million, and the total distance traversed by the trajectories is 9 million kilometers. The average sampling interval is 177 seconds, with an average length of 623 meters.

Table 2: The summary of simulation settings

Parameters	Settings
# of observations in one trajectory ($\mathtt{Num}_o$)	30
# of trajectories ($\mathtt{Num}_\mathsf{T}$)	25K, 50K, 100K, 250K, 500K
Radius of each virtual landmark (r_k^c)	5m, 10m, 15m, 20m, 25m
# of virtual landmark in a query ($\mathtt{Num}_\mathsf{P}$)	20, 25, 30, 35, 40
Query time interval (τ_q)	1h, 2h, 4h, 6h, 8h

• Experimental Settings: There are two metrics for evaluating the performance of the proposed UTM-tree, they are (1) running time and (2) the number of visited trajectories. Table 2 shows all the parameter settings in the experiment. Specifically, for a virtual landmark $\mathsf{P}_k = (x_k^c, y_k^c, r_k^c, t_k^s, t_k^e) \in T_q$, query time interval equals $\tau_q = t_k^e - t_k^s$. Moreover, considering a day with 24 hours, τ_q is randomly selected from 8:00 am to 6:00 pm in the same day. For each P_k, its coordinates (x_k^c, y_k^c) are randomly selected such that $x_k^c \in (116.1650, 116.6201)$ and $y_k^c \in (39.7133, 40, 0070)$. Besides, the maximum speed v_{max} in-between any two continuous observations is set to a random number in $[10\ km/h, 50\ km/h]$.

• Other approaches evaluated: To evaluate the performance of the UTM-tree, we implemented two other approaches for comparison purposes, they are (1) baseline method and (2) temporal-first matching (TF-matching). In the baseline method, we do not index the trajectories at all. More specifically, we evaluate every pair of $(\mathsf{B}_i, \mathsf{P}_k)$ for the *CSQ* query. In the TF-matching method, we follow the steps in [9] to build a temporal filtering tree. Specifically, we split a day into 12 time slots by considering every two hours as a time slot. And then, we store each trajectory into the corresponding nodes. In the TF-matching method, all the trajectories are processed by the temporal filtering tree.

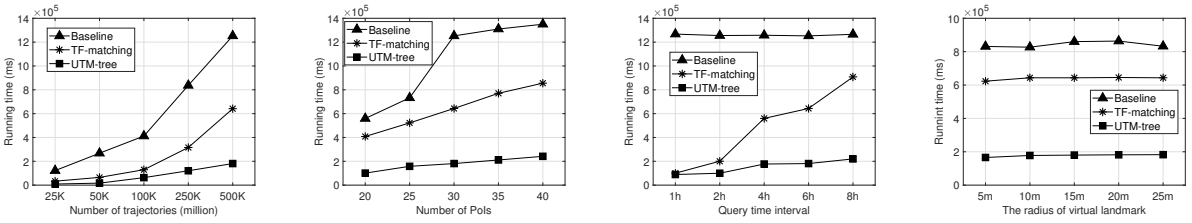
4.2 Performance Evaluation

This section compares the proposed UTM-tree with the baseline and the TF-matching in terms of query running time and the number of visited trajectories. The number of visited trajectories means how many trajectories from the original database still need to be considered after using the UTM-tree for spatial filtering and temporal filtering.

First of all, for the three approaches, the query running time (in Fig. 6 (a)) and the number of visited trajectories (in Fig. 7 (a)) are increasing as $\mathtt{Num}_\mathsf{T}$ increase. Second, we can see that the UTM-tree is much faster than the baseline method and the TF-matching method. For instance, the UTM-tree is 6.9× faster than the baseline method and 3.5× faster than the TF-matching method with 500K trajectories. Third, we found that for the UTM-tree, as $\mathtt{Num}_\mathsf{T}$ becomes larger, the increase for the query running time and the number of visited trajectories is not significant.

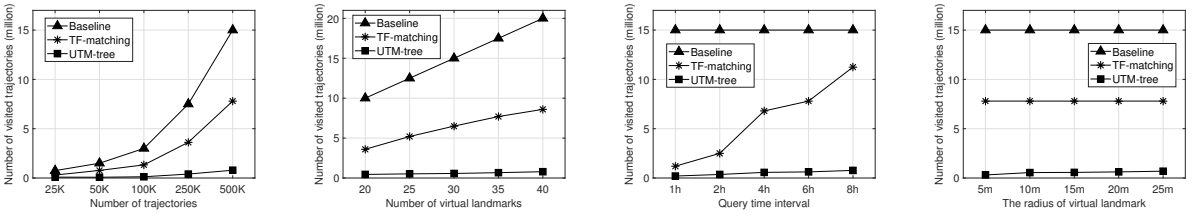
Next, the comparison results for three approaches under the different $\mathtt{Num}_\mathsf{P}$ are shown in both Fig. 6 (b) and Fig. 7 (b). We can see that $\mathtt{Num}_\mathsf{P}$ has the least influence on the performance of the UTM-tree. In contrast, the performance of the baseline method and the TF-matching approach is degraded dramatically.

The influence of τ_q on the three approaches under different experimental settings are shown in both Fig. 6 (c) and 7 (c). The running time and the number of visited trajectories in TF-matching



(a) Query running time vs. $\mathsf{Num_T}$ (b) Query running time vs. $\mathsf{Num_P}$ (c) Query running time vs. τ_q (d) Query running time vs. r_n^c

Figure 6: The comparison of baseline method, TF-matching, and UTM-tree for query running time



(a) # of visited trajectories vs. $\mathsf{Num_T}$ (b) # of visited trajectories vs. $\mathsf{Num_P}$ (c) # of visited trajectories vs. τ_q (d) # of visited trajectories vs. r_n^c

Figure 7: The comparison of baseline method, TF-matching, and UTM-tree for number of visited trajectories

approach increase significantly when τ_q becomes wider. However, the UTM-tree is more stable against the changes in τ_q.

Last, Fig. 6 (d) and 7 (d) compare the performance of the three approaches under different r_k^c. Both the three approaches do not change significantly with different r_k^c. Overall speaking, the UTM-tree is more efficient and stable than the baseline method and the TF-matching approach in terms of running time and the number of visited trajectories under different experiment settings. Specifically, the UTM-tree runs faster than other approaches and is less sensitive to query time intervals, virtual landmark radius, and the number of virtual landmarks in each query.

5 RELATED WORK

(1) *Indexing uncertain trajectories.* There are a number of indexing structures, which have been proposed for uncertain spatio-temporal data, such as UTH [13], Grid-based indexing [3], and UST-tree [5]. However, UTH-tree was used for indexing uncertain trajectories on the road network. UST-tree was used for approximating diamond-based moving objects that follow the Markov-chain model. Grid-based approach was not efficient for indexing beads and necklaces since it is time-consuming to compute the overlapped region between grids and ellipses. Therefore, we devise a novel indexing structure called UTM-tree, which is based on the classic M-tree and is efficient for indexing the uncertain trajectories in the form of beads.

(2) *Queries for uncertain trajectories.* There are many studies that were proposed for querying uncertain trajectories, such as spatio-temporal similarity join [9], semantic similarity join [3], nearest-neighbor queries [11], and top-k similarity query [6]. However, most of them retrieved qualified trajectories based on some spatial/temporal criteria. None of the previous works study the problem of contact similarity in terms of spatial intersection and longest contact time duration among trajectories. This work is the first research to formally define the problem of contact similarity and propose the corresponding *CSQ* for the problem.

6 CONCLUSION

In this work, we have formally defined the concept of contact similarity and proposed a novel query, called *CSQ*. Next, we designed a novel indexing structure called UTM-tree, for managing and querying uncertain trajectories. Besides, we conducted extensive experiments on the Beijing Taxi dataset. The experimental results demonstrated the efficiency and stability of the UTM-tree on *CSQ*. There are many interesting future directions, e.g., (1) contact modeling between MOs on road networks, (2) indoor contact modeling between MOs, (3) *CSQ* in terms of contact frequency, (4) second-generation contact between MOs, and (5) performance evaluation with more real-world datasets.

REFERENCES

[1] Prithu Banerjee, Sayan Ranu, and Sriram Raghavan. 2014. Inferring uncertain trajectories from partial observations. In *2014 IEEE ICDM*. IEEE, 30–39.

[2] Ionut Cardei, Cong Liu, Jie Wu, and Quan Yuan. 2008. DTN routing with probabilistic trajectory prediction. In *WASA*. Springer, 40–51.

[3] Lisi Chen, Shuo Shang, Christian S Jensen, Bin Yao, and Panos Kalnis. 2020. Parallel Semantic Trajectory Similarity Join. In *2020 IEEE 36th ICDE*. 997–1008.

[4] Paolo Ciaccia, Marco Patella, Fausto Rabitti, and Pavel Zezula. 1997. Indexing metric spaces with m-tree. In *SEBD*, Vol. 97. 67–86.

[5] Tobias Emrich, Hans-Peter Kriegel, Nikos Mamoulis, Matthias Renz, and Andreas Züfle. 2012. Indexing uncertain spatio-temporal data. In *Proceedings of the 21st ACM CIKM*. 395–404.

[6] C. Ma, H. Lu, L. Shou, and G. Chen. 2013. KSQ: Top-k Similarity Query on Uncertain Trajectories. *IEEE TKDE* 25, 9 (2013), 2049–2062.

[7] Mark EJ Newman. 2002. Spread of epidemic disease on networks. *Physical review E* 66, 1 (2002), 016128.

[8] Dieter Pfoser and Christian S Jensen. 1999. Capturing the uncertainty of moving-object representations. In *SSD*. Springer, 111–131.

[9] Shuo Shang, Lisi Chen, Zhewei Wei, Kai Zheng, and Panos Kalnis. 2017. Trajectory similarity join in spatial networks. *VLDB Endowment* (2017).

[10] Goce Trajcevski, Alok Choudhary, Ouri Wolfson, Li Ye, and Gang Li. 2010. Uncertain range queries for necklaces. In *2010 11th MDM*. IEEE, 199–208.

[11] Goce Trajcevski, Roberto Tamassia, Hui Ding, Peter Scheuermann, and Isabel F Cruz. 2009. Continuous probabilistic nearest-neighbor queries for uncertain trajectories. In *Proceedings of the 12th EDBT*. 874–885.

[12] Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. 2011. Driving with knowledge from the physical world. In *Proceedings of the 17th ACM SIGKDD*. 316–324.

[13] Kai Zheng, Goce Trajcevski, Xiaofang Zhou, and Peter Scheuermann. 2011. Probabilistic range queries for uncertain trajectories on road networks. In *Proceedings of the 14th EDBT*. 283–294.

Short Paper

AdCom: Adaptive Combiner for Streaming Aggregations

Felipe Gutierrez[1] Kaustubh Beedkar[1] Abel Souza[2] Volker Markl[1]
[1]Technische Universität Berlin, Germany [2]Umeå University, Sweden
firstname.lastname@tu-berlin.de firstname.lastname@cs.umu.se

ABSTRACT

Continuous applications such as device monitoring and anomaly detection often require real-time aggregated statistics over unbounded data streams. While existing stream processing systems such as Flink, Spark, and Storm support processing of streaming aggregations, their optimizations are limited with respect to the dynamic nature of the data, and therefore are suboptimal when the workload changes and/or when there is data skew. In this paper we present AdCom, which is an adaptive combiner for stream processing engines. The use of AdCom in aggregation queries enables pre-aggregating tuples upstream (i.e., before data shuffling) followed by global aggregation downstream. In contrast to existing approaches, AdCom can automatically adjust the number of tuples to pre-aggregate depending on the data rate and available network. Our experimental study using real-world streaming workloads shows that using AdCom leads to 2.5–9× higher sustainable throughput without compromising latency.

1 INTRODUCTION

Continuous or real-time applications often require real-time aggregated statistics over an unbounded stream of events or tuples. For example, ride-sharing platforms such as Uber and Lyft utilize real-time aggregated statistics about traffic conditions to provide suggestions on trip routes [1, 27]. As another example, interactive entertainment platforms such as King.com provide their data science teams with real-time aggregated statistics over billions of user events from different games and systems [11].

To efficiently process continuous streams of data in real-time, applications rely on Distributed Stream Processing Engines (SPEs) such as Spark Streaming [29], Apache Flink [5], Apache Samza [19], or Apache Storm [24]. These systems enable data stream processing with low-latency and high throughput, and can scale by distributing computations among a cluster of machines.

In the particular case of *streaming aggregations*, which are `groupBy-aggregation` queries on continuous data, query execution involves shuffling of data stream tuples among compute machines of the cluster. This data shuffling involves communication between machines via network, which incurs a performance overhead in terms of a decrease in throughput and an increase in end-to-end latency. The shuffling overhead—of which the network bandwidth between machines is an important factor—also increases as the degree of parallel processing increases. Therefore, it is essential to reduce the shuffling overhead to improve the overall performance of SPEs for streaming aggregations.

Current SPEs optimize data shuffling by extending the "combine plus reduce" pattern of MapReduce (batch) to streaming. In particular, SPEs first pre-aggregate upstream (i.e., locally aggregate tuples in each partition before shuffling over the network) and then perform a global aggregation operation downstream. To cater to the needs of continuous applications, the number of input tuples accumulated prior to applying local aggregation is based on a pre-configured mini-batch interval (e.g., for every 1000 tuples or every 5 seconds).

While SPEs allow configuring the mini-batch interval, its fixed size during query execution is not ideal for real-time applications. This is because, it lacks adaptability to the dynamic nature (w.r.t. data rate changes and/or data skew) of datastreams. For example, if the throughput of a data stream suddenly increases to what the SPE cannot sustain (i.e., when the network gets saturated), SPE inflicts a "backpressure" on the upstream operators. Backpressure leads to an increase in the system's latency. Likewise, if there is (sudden) data skew in the stream, some instances of the (downstream) aggregation operation will process more records than others leading to saturation of network buffers and consequently affecting latency. One way to deal with the unpredictable nature of datastreams is to re-configure the size of the network buffers and/or mini-batch intervals to pre-aggregate. However, this requires re-starting the query, which is expensive and undesirable for continuous applications.

In this paper, we propose AdCom, which is an adaptive combiner for SPEs. In contrast to pre-aggregating tuples based on a fixed mini-batch interval, AdCom uses dynamic mini-batch intervals. This allows "on-the-fly" adjusting of the number of tuples to pre-aggregate. To deal with sudden changes in data rate, AdCom utilizes a feedback mechanism consisting of a proportional-integral controller that continuously monitors its network buffers, and an actuator that signals AdCom to adjusts its mini-batch interval. Thus, a high network usage results in pre-aggregating more number of tuples and vice-versa. This allows SPEs to adapt to sudden data rate changes and/or skew, and achieve a higher sustainable throughput without compromising latency.

We implemented AdCom in Apache Flink[1], which we considered as a representative SPE, and performed an extensive evaluation using real-world and synthetic datasets. Our results indicate that with AdCom, Flink can autonomously adapt to data rate changes and can execute aggregation queries with higher sustainable throughput (up to 2.5×) compared to existing approaches.

2 BACKGROUND

We start with a discussion on streaming aggregations that we consider in this paper. We then describe the limitations of SPEs in execution aggregation queries. We use Apache Flink as a representative SPE to explain key concepts. These concepts also generalize to other SPEs like Spark Streaming or Apache Storm.

2.1 Streaming Aggregations

We focus on the computing of streaming aggregates that are most common in stream analytics applications. More specifically, we consider unbounded aggregations on continuous queries that have the following general form:

```
dataStream.groupBy(...).aggregate(...)
```

In the above general form, the `groupBy(...)` transformation first groups elements of the data stream by the specified key(s).

[1]https://github.com/TU-Berlin-DIMA/AdCom

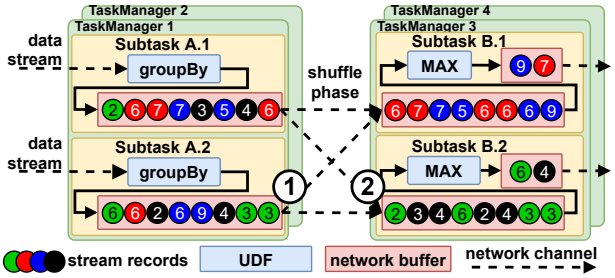


Figure 1: Illustration of executing a `groupBy-max` query.

Then, the `aggregate(...)` transformation computes a "rolling" aggregate as an output data stream. Such queries are the backbone of many common data analytic tasks such as retrieving, gathering, and organizing data. Common aggregate functions supported by SPEs include `sum()`, `min()`, `max()`, and `avg()` among others. SPEs also support parameterizing aggregate transformations with User Defined Functions (UDFs).

As an example, consider a weather analytics dashboard that continuously updates the maximum temperature of all regions in a neighborhood. The application receives as an input, a data stream S of `<timeStamp,regionId,temperature>` events, and executes the aggregation query[2]:

`S.groupBy(regionId).max(temperature)`

For a given stream such as:

$$[1, A, 23], [2, A, 25], [1, B, 19], [1, C, 28], [2, B, 18], \ldots$$

the aggregation query produces the following resulting stream:

$$[A, 23], [A, 25], [B, 19], [C, 28], [B, 19], \ldots$$

Figure 1 gives a high level overview of executing such an aggregation query on a Flink cluster. It includes two tasks A and B with their respective subtasks running in parallel. In this example, subtasks A.1 and A.2, which perform the `groupBy` transformation, communicate with the two instances of task B, which perform the `max()` transformation. The `groupBy()` operations leads to shuffling of stream elements over the network[3]. Before shuffling, the output of the upstream (or sender) task is first queued in a (network) buffer at ①. The events, which are already grouped by key, are then flushed on to the network after a pre-configured timeout (e.g., 100ms) or when the buffers are full. Likewise, on the receiver side, the data is first queued in a buffer at ②, which is then consumed by the downstream subtasks.

2.2 Limitations of SPEs

SPEs strive to achieve a high sustainable throughput and low end-to-end latency. In real-world workloads, however, SPEs have to deal with two scenarios that affect its throughput and latency: (1) When the data stream's arrival rate increases to what a system can not handle, and (2) when there is data skew, which causes some instances of the aggregation tasks to process many more records than others. In both these scenarios, SPEs exhibit a backpressure mechanism, where the stream of events is queued up in network buffers before being processed. This leads to an increase in end-to-end latency, and in the worst case stalls the dataflow.

In the case of aggregation queries, backpressure on the sender tasks, either due to a high arrival rate or data skew, can be mitigated to some extent by first locally aggregating. This is akin to the use of a combiner in MapReduce [8]. In the context of data

[2]for brevity, we suppress the `map()` (or `project()`) transformation to project out the `timeStamp` attribute

[3]Other similar transformation are `keyBy()` and `reduceByKey()`.

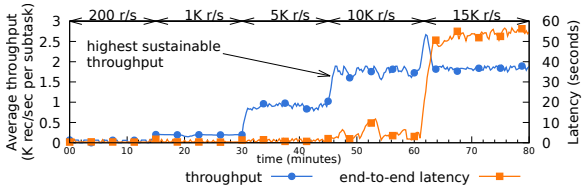


Figure 2: Limitations of SPE w.r.t. data rate changes when executing a `groupBy-sum` query.

streaming, SPEs pre-aggregate records based on small mini-batch interval (e.g., for every 5 seconds) before shuffling them over the network. However, this pre-aggregation is suboptimal as using a fixed mini-batch interval leads to an increased end-to-end latency when workload changes and/or when there is skew.

We illustrate this limitation with an experiment that we performed using Apache Flink. We considered a `groupBy-sum` query and a setup comprising four local (pre-) aggregation tasks and four global aggregation tasks. The mini-batch interval to pre-aggregate was set to 5s and 5K records. Figure 2 shows the system's throughput (in blue; left axis) and its end-to-end latency (in orange; right axis).

As we observe in Figure 2, the end-to-end latency remains constant (~0.5s) as the data rate increases from 200 records/s to 5K records/s. Also, the system achieves its highest sustainable throughput of ~8K records/sec. Note that when the arrival rate further increases to 15K records/s, we see a spike in end-to-end latency. This is because, the system exhibits a backpressure mechanism and consequently an increased (up to 60s) end-to-end latency when it can no longer cope with the arrival rate.

One way to deal with the changes in workload is to re-configure the number of tuples to pre-aggregate and/or the size of the network buffers. However, this requires restarting the query each time the workload changes, which is expensive and undesirable for real-time applications.

3 ADAPTIVE COMBINER

We now present `AdCom`, an adaptive combiner for SPEs that overcomes the limitations mentioned above. The use of `AdCom` aids in optimizing aggregation queries in SPEs by dynamically adjusting the mini-batch interval to pre-aggregate prior to data shuffling, which enables the SPE to constantly achieve the highest sustainable throughput without compromising latency.

The key challenge in designing `AdCom` is to determine when to emit locally aggregated tuples. As discussed above, using a fixed mini-batch interval is useful, but when the arrival rate of records is higher than what the network can handle, the backpressure mechanism is activated and leads to high latency. On the other hand, if the arrival rate is too low, then a large mini-batch interval will diminish the lowest achievable latency for the query.

3.1 The Feedback Mechanism

We tackle the above challenge by making use of a feedback mechanism, which comprises (1) a controller and (2) an actuator. The controller continuously monitors the network buffers and computes an optimal mini-batch interval for the current workload. The actuator periodically sends "signals" (i.e., the new configuration for the mini-batch interval) to the parallel instances of `AdCom`. This feedback mechanism allows `AdCom` to dynamically adjust the time (and/or number of tuples) for which it should locally aggregate tuples.

Figure 3 gives an overview of executing an aggregate query using AdCom. Before we delve into its details, recall that backpressure is inflicted when the incoming arrival rate of records surpasses what the system can handle. In the context of query execution, this happens when the (sender's) network buffer is queued up with records that it cannot flush to the network (i.e., shuffle). More formally, denote by λ the rate at which a pre-aggregation task writes records to its buffer (i.e., the mini-batch interval) and by μ the rate at which the records are flushed out from the buffer. Further, let $\rho = \frac{\lambda}{\mu}$ denote the fraction of buffer utilized. Intuitively, a backpressure situation arises when $\rho \geq 1$. The key idea of our approach is to always keep $\rho < 1$ by continuously updating λ, which translates to adapting the mini-batch interval to pre-aggregate[4].

3.2 AdCom's Controller and Actuator

We now detail the working of the controller and actuator. As shown in Figure 3, the controller and the actuator are part of Flink's job manager. This allows us to globally control all (parallel) AdCom instances (i.e., pre-aggregation subtasks) that are executed in the same phase. This is crucial for preserving the query semantics with respect to the order in which the records are processed downstream. Otherwise, having a controller for each of AdCom's parallel instance would result in writing records (to its buffers) at different time intervals. This may lead to a change in the order that records are processed downstream, which may be undesirable depending on the application.

We use a proportional-integral (PI) controller to continuously compute the optimal λ. PI controllers have been widely used in control systems and applications that require continuous modulated control[5]. We refer interested readers to [12] for a comprehensive overview. In what follows, we discuss how we use a PI controller in the context of performing streaming aggregations.

At high level, and as illustrated in Figure 3, the controller continuously calculates an error value $e(t)$ for each time t, which is the difference between the desired value ρ_d of the network buffer utilization and the measured current buffer utilization ρ. Based on the error value, it updates λ based on a proportional (K_P) and integral (K_I) terms. The proportional control term determines the correction in λ, which is proportional to $e(t)$. The integral term further applies a correction based on past values of $e(t)$, to diminish the residual error (i.e., when $e(t) > 0$ or $e(t) < 0$ after applying the proportional correction).

In more detail, we compute $e(t)$ as follows. Denote by $A_1, \ldots, A_n$ the parallel instances of AdCom, and by ρ_{A_i} the buffer utilization for instance A_i. Since we want to globally control all instances (as mentioned above), we compute the error value as:

$$e(t) = -\begin{cases} \rho_d - \text{avg}(\rho_{A_1}, \ldots, \rho_{A_n}) \text{ if } \nexists A_i \text{ s.t. } \rho_{A_i=1} \\ \rho_d - 1 \text{ otherwise} \end{cases}$$

In other words, we compute $e(t)$ using the average buffer utilization across AdCom's parallel instances. To cope with data skew, we compute $e(t)$ as $\rho_d - 1$, i.e., when there is at least one AdCom instance with 100% buffer utilization. The desired value ρ_d is set based on the Service-Level Objective (SLO) to keep low or medium backpressure on the upstream operators, which usually corresponds to 50–80% usage of the network buffers.

When $e(t) > 0$, the ρ signals are too high, then the controller produces an input signal that decreases ρ. This materializes by

[4]Alternatively, one can continuously update μ or both.

[5]Some control systems additionally make use of a derivative component, i.e., a PID controller; we do not use it as it is suitable only for slow moving loops.

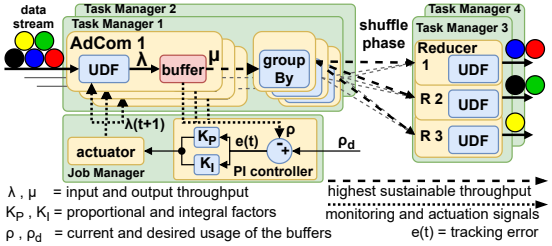


Figure 3: Overview of AdCom.

increasing the pre-aggregating time (and decreasing λ) at AdCom. Vice-versa, when $e(t) < 0$, the controller produces an input signal that increases the usage of the buffers. This leads to "less aggregation" and increase in λ at AdCom. Finally, based on the $e(t)$, the controller applies a correction based on proportional (K_P) and integral (K_I) terms, which we explain next.

Proportional Factor (K_P). It is crucial for any controller to let the magnitude of the corrective action depend upon the magnitude of the error. This states how quickly AdCom responds to errors. For instance, small errors lead to small adjustments, whereas larger errors result in greater corrective actions. This is accomplished by the proportional factor K_P (Equation 1). It helps the controller apply the largest control action if $e(t)$ is large and not making the system unstable if $e(t)$ is small. The next proportional parameter (i.e., the mini-batch interval) to pre-aggregate records is denoted by $\lambda(t+1)$ and it is computed based on the current parameter $\lambda(t)$ summed with the proportional factor K_P times the error $t(e)$, as follows:

$$\lambda(t+1) = \lambda(t) + [K_P.e(t)] \quad (1)$$

The controller estimates a proportional correction if $e(t)$ is within the desired min and max values of ρ_d. However, since we consider average buffer utilization, it may be possible that $e(t)$ becomes zero. Therefore, no correction is taken in this case. One way to fix this is to reduce the desired range or increase K_P. However, this leads to AdCom making rapid and unstable adjustments, which is very undesirable. Therefore, we further use an integral factor K_I to determine the corrective action.

Integral Factor (K_I). The integral factor makes the controller not react only based on the momentary $e(t)$ but also based on its previous values. It provides a way to amplify small errors and keep adding them up over time. The accumulated values provide a significant control signal and help the system not oscillate when it reaches the optimal time to locally aggregate tuples. We compute the integral factor based the three last values of $e(t)$ with a sliding window-based histogram that implements the reservoir algorithm. In our experiments, this window size was enough to achieve a steady-state on the pre-aggregation parameter of AdCom (see Section 4). Equation 2 composes the AdCom' *PI controller* with its proportional and integral factors that we use to calculate the new pre-aggregation parameter λ.

$$\lambda(t+1) = [\lambda(t) + (K_P.e(t))] + [\lambda(t) + (K_I.\int_{t}^{(t-3)} e(\tau).d\tau)] \quad (2)$$

3.3 Using AdCom in Flink

One can make use of AdCom at the API level, by parameterizing it with the `groupBy()` key and a query specific UDF for pre-aggregation. For example, our modifications to Flink allow us to write the example aggregate query of Section 2 as:

```
S.adcom(regionId, max(temperature))
 .groupBy(regionId).max(temperature)
```

4 EXPERIMENTS

We conducted an experimental study using a real-world dataset in the context of a ride-sharing application. Our goal was to compare the performance of Flink when using AdCom and other existing optimizations. In particular, we investigated: (i) how well AdCom adapts when data rate changes; (ii) how well AdCom fares when there is data skew; (iii) AdCom's resource efficiency, and; (iv) how distributive and user-defined aggregate functions affect its performance. We found that:

- Flink+AdCom achieved up to 2.5–9× higher sustainable throughput when data rate increased by 4×.
- AdCom is competitive to state-of-the-art when handling skew.
- Flink+AdCom with 8 reducers achieved 9× higher throughput than Flink+noOptimization, and 3× higher throughput than state-of-the-art.
- Flink+AdCom achieved similar throughput but lower latency when using distributive and user-defined aggregate functions.

4.1 Experimental Setup

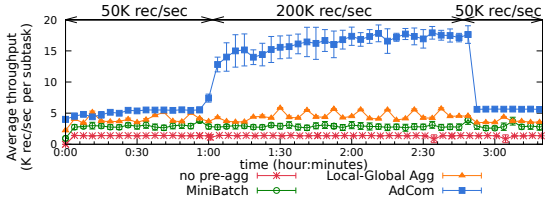
Implementation and cluster. We implemented AdCom as a new operator in Apache Flink. The PI controller and the actuator are implemented as components of Flink's job manager and use Flink's job monitoring API to monitor network buffer usage of AdCom. All of our experiments were run on a local Flink cluster consisting of four machines, each with 16GB of main memory, one hard disk of 2TB, one Intel Xeon 2.66GHz 64-bit 8 core CPU, and Ubuntu 16.04.6 (kernel GNU/Linux 4.4.0) as the operating system. The machines in the cluster are connected via 1 Gbps Ethernet. We used Apache Flink 1.11.2 and Java 1.8 for our implementation. We configured the Flink cluster with four Task Managers and one Job Manager, and set the maximum sub-task parallelism to 8 per node (i.e., the same number of CPU cores).

Datasets and aggregation queries. We used the New York City Taxi and Limousine Commission (TLC) [22] dataset. It consists of three million taxi trip records with fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. We additionally considered the TPC-H benchmark dataset [4] with scale factor 5.

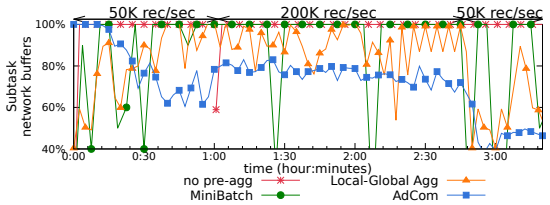
For the TLC dataset, we considered queries with algebraic and distributive aggregate functions. In particular: (i) we considered an event-count query (which we denote by Q1 in the text below) that sums the number of passengers for each taxi driver, and (ii) and an aggregation query that computes for each taxi driver the average number of passengers, trip distance, and trip time (denoted by Q2). For the TPC-H data we considered the TPC-H query 1, which is an aggregation query over a single table (lineitem) consisting of COUNT, SUM, and AVG aggregate functions.

We followed Karimov et al. [15] to generate on-the-fly data streams for benchmarking streaming applications. In particular, we load the datasets in memory and implement a streaming data source that can emit events with different characteristics (see Sections 4.2 and 4.3 below).

Baseline. To evaluate the performance of AdCom, we compare it with no pre-aggregation, and with the state-of-the-art streaming aggregation optimizations available in Flink [13]. In more detail, we compare against (i) MiniBatch, which buffers input records before the shuffle phase, and (ii) Local-Global Aggregation, which executes the MiniBatch and then pre-aggregates records locally (akin to Combine plus Reduce in MapReduce).



(a) Throughput



(b) Network buffer usage

Figure 4: Effect of change in data rate.

We set the maximum latency and the maximum number of input records that can be used for MiniBatch and Local-Global aggregation to 3 seconds and 3K events, respectively.

4.2 Effect of Change in Data Rate

We first study how well Flink adapts to a sudden surge in data arrival rate when using AdCom and compare its performance with the baseline approaches. We consider the TLC dataset, and a stream data source that for the first 60 mins generates events at 50K records/sec, then for the next 100 mins at 200K records/sec, and finally again reverts to a rate of 50K records/sec. We considered the aggregation query Q1 and measured its throughput and latency, for which the results are shown in Figure 4.

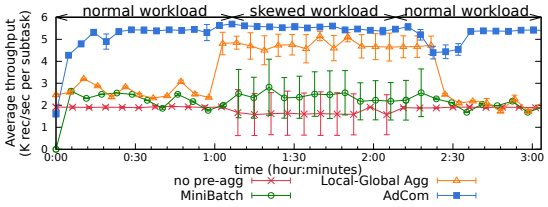
We observed that Flink+AdCom achieved a higher throughput than Flink+MiniBatch and Flink+LocalGlobal (see Figure 4(a)). More specifically, when the input throughput is 50K rec/sec, AdCom achieved an optimal mini-batch interval of 3.5s (starting from the default value of 500ms) compared to a fixed interval (of 3s and 3K events) for Flink+LocalGlobal. This allows us to obtain a slightly higher throughput for the first 60mins. As the input rate surges to 200K records/sec, AdCom further adapts its mini-batch interval (from 3.5s to 8.5s), which allows Flink+AdCom to achieve much higher throughput (from 3.6× up to 9×).

Figure 4(b) also shows the percentage of Flink's network buffer utilization for such a workload, which correlates to its latency. High buffer utilization indicates high backpressure and increased latency, whereas a lower buffer utilization indicates a low backpressure and lower latency. We observed that with AdCom, Flink had an overall lower buffer utilization due to AdCom constantly adjusting its mini-batch interval. When the workload suddenly surges to 200K records/sec, AdCom had a buffer utilization (up to 80%) but stabilizes after it reaches a sustainable throughput of 18K records/sec per subtask (see Figure 4(a)) pre-aggregating records every 8.5s. With baseline approaches, we observe that pre-configured mini-batch intervals are not ideal for dynamic workloads and lead to high buffer usage with surges in data rate.

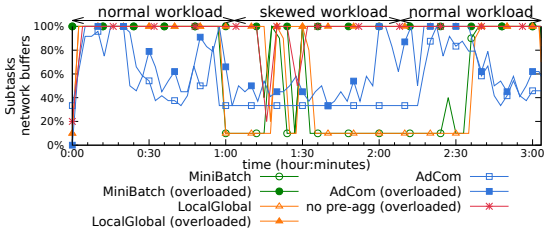
Our results indicate that AdCom allows Flink to adapt to data rate changes and achieve a higher sustainable throughput.

4.3 Effect of Skew Workloads

We now proceed to evaluate how well Flink performs in the presence of skew workloads. We consider the TLC dataset, and a stream data source at 50K records/sec that for the first 60 mins



(a) Throughput



(b) Network buffer usage

Figure 5: Effect of data skew.

generates tuples following a uniform key distribution, then for the next 70 mins following a skewed distribution, and finally again reverts to a normal distribution. We considered the aggregation query Q1 and measured its throughput and Flink's network buffer utilization, for which the results are shown in Figure 5.

We observed that the LocalGlobal optimization allowed Flink to reduce the network shuffle and cost of global aggregation in the presence of skew. However, as shown in Figure 5(a), we observed that throughput across subtasks had some variation. Flink+AdCom also responded well to skew as it also first locally aggregates, and achieved a slightly higher throughput with negligible variation. This is because, AdCom adjusted its mini-batch interval to 3.5s compared to (pre-configured) 3s of LocalGlobal. We also show the network buffer utilization for this setting in Figure 5(b). Note that different subtasks have different buffer utilization, and hence we show two lines (one for the overloaded task and other for remaining tasks). As observed in Figure 5(b), AdCom leads to lower buffer utilization and hence lower backpressure.

Overall, our experiments indicate that AdCom is competitive to LocalGlobal aggregation in dealing with data skew.

4.4 Resource Efficiency

We now evaluate the resource efficiency of Flink when using AdCom for data streams with high arrival rate. A common strategy to cope with high arrival rate is to add more resources, i.e., increase the degree of parallelism. When we add more global (i.e.: reducer) subtasks after the shuffle phase, it helps the query to avoid backpressure. Hence, it can sustain a higher throughput.

We consider the TLC dataset and a streaming source that generates events at a fixed rate of 200K rec/sec and process query Q1 with different aggregation optimizations. To deal with a higher workload, we set the maximum latency and the maximum number of input records that can be used for MiniBatch and LocalGlobal aggregation to 7 seconds and 7K events, respectively.

Figure 6 shows the throughput of the query for different strategies. We observed that Flink+AdCom with 8 reducers achieved 9× higher throughput than Flink+noOptimization, and 3× higher throughput than Flink+MiniBatch and Flink+LocalGlobal. Further, Flink+AdCom required only 8 reducers to achieve the same throughput that Flink+LocalGlobal achieves with 24 reducers. This is because the MiniBatch and the LocalGlobal approaches

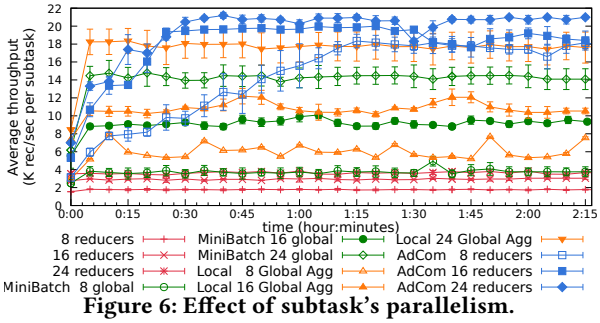


Figure 6: Effect of subtask's parallelism.

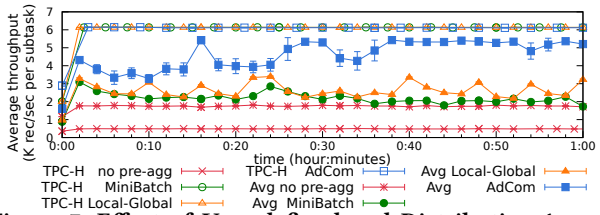


Figure 7: Effect of User-defined and Distributive Aggregate Functions.

first accumulates (mini-batches) records every 7 seconds, and then aggregate every 7K tuples. In contrast, AdCom leverages a hash-based data structure, which allows us to consume and pre-aggregate tuples in a single pass and hence requires less resources.

Based on these experiments, we conclude that AdCom is more resource efficient compared to MiniBatch and LocalGlobal.

4.5 Results for User-defined and Distributive Aggregate Functions

Lastly, we evaluate the performance of AdCom for aggregation queries with both distributive and algebraic aggregate functions. We considered queries Q2 (on the TLC dataset) and TPC-H Q1 and set the maximum latency and the maximum number of input records that can be used for MiniBatch and LocalGlobal aggregation to 1s and 1K events, respectively. The arrival rate for each stream was 50K records/sec. The results are shown in Figure 7.

We first discuss the results for TPC-H query Q1. We observed that Flink achieves the same throughput when using either of AdCom, MiniBatch, or LocalGlobal. This is because the configured maximum latency of 1s and the maximum number of input records of 1K events that configured for MiniBatch and LocalGlobal aggregation was sufficient to handle the workload and the user-defined aggregate function. Although Flink+AdCom achieved a similar throughput, it adapts its mini-batch interval from 1s to 50ms, and thus achieving a lower latency than Flink+MiniBach and Flink+LocalGlobal (not shown here).

For query Q2, which involves an algebraic aggregate, we observed that Flink+AdCom again achieves a higher throughput than Flink+MiniBatch and Flink+LocalGlobal. We note that, other than AVG(), Flink+LocalGlobal cannot optimize queries with algebraic aggregate functions. In contrast, AdCom allows specifying any UDF for pre-aggregating which makes it suitable to aggregation queries that accept a "useful" combiner.

Overall, we found that Flink+AdCom achieves a higher throughput without compromising on latency even for distributive aggregate functions. We also performed experiments (not shown here due to space constraints) considering queries involving other distributive and algebraic aggregate functions such (e.g., sum, max, min, and top-k) and observed similar performance.

5 RELATED WORK

We now discuss how ideas presented in this paper are related to prior work on adaptive stream processing. Due to space constraints, we discuss works that are most related.

Das et al. [7] studied the effect of batch sizes on throughput and latency, and proposed a control algorithm based on fixed point iteration. Their work focuses on determining the optimal batch size for ingesting the data into SPE and relies on past queries. Zhang et al. [31] proposed Dynamic Block and Batch Sizing (DyBBS) using an online control algorithm integrated with isotonic regression. Besides adjusting the micro-batch size, DyBBS also adjusts the execution parallelism (i.e., batch size/block size in Spark). Instead of adapting the batch size, Drizzle [26] focuses on optimally scheduling multiple batches (or a group), and automatically tunes the group size. A-scheduler [6] further extended this approach by dynamically changing the batch sizes using an expert fuzzy control mechanism. Wu et al. [28] studied the impact of batch size on Kafka streams that are then ingested into SPEs, and proposed a reactive batching strategy to cope with variable network conditions. Lohrmann et al. [17] proposed strategies to switch between adaptive buffer (re-)sizing and dynamic task chaining to optimize execution plans. All these approaches focus on adjusting the batch size w.r.t. the entire query or prior to ingesting the data stream into the SPE. In contrast, we focus on the batch size specific to an (aggregation) operator. `AdCom` is thus complementary to the above approaches.

Apart from dynamically configuring the batch size, many works have also focused on adapting the execution plan. WASP [14] uses a network-aware framework that is able to adapt the query plan to the resources available in run-time via task re-assignment, operator scaling, and query re-planning. Nasir et al. [18] proposed a hash-based algorithm to partition data that optimizes network shuffle and deals with skew workloads. While [3, 16, 20, 25] have studied eliminating redundant computations via dynamically sharing of data and/or compute, [10, 21] focused on adaptability via query complication. Eddies [2] also tackled the problem of unpredictable workloads by reordering operators at runtime.

Feedback mechanisms have been a central component in enabling adaptive stream processing. FAST [9] uses adaptive sampling methods based on a PID controller to adjust the sampling rate for processing streams. Tolosana-Calasanz et al. [23] uses a feedback mechanism based solely on proportional factor to minimize resource utilization. In `AdCom`, we leverage a PI controller, which is specific to optimizing local-aggregations.

6 CONCLUSION AND FUTURE WORK

Adaptability of SPEs to varying workloads is important for real-time applications. In this paper, we considered streaming aggregations, which are the backbone of many real-time applications. We have proposed `AdCom`, an adaptive combiner for SPEs, which improves the performance of aggregation queries under variable workloads. We have proposed a lightweight feedback mechanism that continuously monitors the network buffers, and allows `AdCom` to autonomously adapt to varying workloads. In our experimental evaluation using a real-world dataset, we have shown that `AdCom` achieves a higher sustainable throughput (up to 9×) without compromising latency, is resilient to data skew, and is resource efficient when compared to existing optimizations.

As future work, we plan to extend our work to fog and edge computing environments [30]. In particular, we plan to study how to adapt `AdCom` for nodes that are resource (e.g., memory) constrained. We also plan to make `AdCom` adaptive to hybrid environments that include unreliable compute nodes and network channels. Lastly, we plan to extend existing optimizers to create the `AdCom` UDF during compile time, so it can relieve the developers of this task.

ACKNOWLEDGMENTS

This work was supported by the FogGuru project, which has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452, and by the German Ministry for Education and Research as BIFOLD - "Berlin Institute for the Foundations of Learning and Data" (01IS18025A and 01IS18037A).

REFERENCES

[1] Introducing AthenaX. 2017. Uber Engineerings Open Source Streaming Analytics Platform. https://eng.uber.com/athenax
[2] Ron Avnuret al. 2000. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*. 261–272.
[3] Shivnath Babuet al. 2004. StreaMon: An Adaptive Engine for Stream Query Processing. In *SIGMOD*. 931–932.
[4] Transaction Processing Performance C. 1988-2021. *TPC-H, a decision support benchmark*. http://www.tpc.org/tpch
[5] Paris Carbone et al. 2015. Apache flink: Stream and batch processing in a single engine. In *IEEE CS*. 28–38.
[6] D. Cheng et al. 2018. Adaptive Scheduling Parallel Jobs with Dynamic Batching in Spark Streaming. In *IEEE TPDS*. 2672–2685.
[7] Tathagata Das et al. 2014. Adaptive Stream Processing Using Dynamic Batch Sizing. In *SOCC*. 1–13.
[8] Jeffrey Deanet al. 2008. MapReduce: Simplified Data Processing on Large Clusters. In *Commun. ACM*. 107–113.
[9] Liyue Fan et al. 2013. FAST: Differentially Private Real-Time Aggregate Monitor with Filtering and Adaptive Sampling. In *SIGMOD*. 1065–1068.
[10] Philipp M. Grulich et al. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *SIGMOD*. 2487–2503.
[11] F. Gyula et al. 2017. Scalable real-time analytics. US Patent App. 15/475,913.
[12] Joseph L Hellerstein et al. 2004. Feedback control of computing systems. In *Wiley Online Library*.
[13] Streaming Aggregation in Flink Table API. 2019. . https://ci.apache.org/projects/flink/flink-docs-stable/dev/table/tuning/streaming_aggregation_optimization.html
[14] Albert Jonathan et al. 2020. WASP: Wide-Area Adaptive Stream Processing. In *Middleware*. 221–235.
[15] J. Karimov et al. 2018. Benchmarking Distributed Stream Data Processing Systems. In *ICDE*. 1507–1518.
[16] Jeyhun Karimov et al. 2019. AStream: Ad-hoc Shared Stream Processing. In *SIGMOD*. 607–622.
[17] Björn Lohrmann et al. 2014. Nephele streaming: stream processing under QoS constraints at scale. In *Cluster computing*. 61–78.
[18] M. A. U. Nasir et al. 2015. The power of both choices: Practical load balancing for distributed stream processing engines. In *IEEE ICDE*. 137–148.
[19] Shadi A. Noghabi et al. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. In *VLDB Endowment*. 1634–1645.
[20] Do Le Quoc et al. 2018. ApproxJoin: Approximate Distributed Joins. In *SoCC*. 426–438.
[21] Filippo Schiavio et al. 2020. Dynamic Speculative Optimizations for SQL Compilation in Apache Spark. In *VLDB Endowment*. 754–767.
[22] New York City Taxiet al. 2020. . https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[23] Rafael Tolosana-Calasanz et al. 2016. Feedback-control & queueing theory-based resource management for streaming applications. In *IEEE TPDS*. 1061–1075.
[24] Ankit Toshniwal et al. 2014. Storm@Twitter. In *SIGMOD*. 147–156.
[25] Jonas Traub et al. 2019. Efficient Window Aggregation with General Stream Slicing.. In *EDBT*. 97–108.
[26] Shivaram Venkataraman et al. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *SOSP*. 374–389.
[27] Alex Woodie. 2019. *What's Behind Lyft's Choices in Big Data Tech*. https://www.datanami.com/2019/05/10/whats-behind-lyfts-choices-in-big-data-tech/
[28] H. Wu et al. 2020. A Reactive Batching Strategy of Apache Kafka for Reliable Stream Processing in Real-time. In *ISSRE*. 207–217.
[29] Matei Zaharia et al. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*. 423–438.
[30] Steffen Zeuch et al. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *CIDR*.
[31] Q. Zhang et al. 2016. Adaptive Block and Batch Sizing for Batched Stream Processing System. In *IEEE ICAC*. 35–44.

Short Paper

Querying Top-k Dominant Traffic Flows on Large Urban Road Networks

Stella Maropaki*
Norwegian University of Science and Technology
stella.maropaki@ntnu.no

Paolo Sottovia
Huawei Munich Research Center
paolo.sottovia@huawei.com

Stefano Bortoli
Huawei Munich Research Center
stefano.bortoli@huawei.com

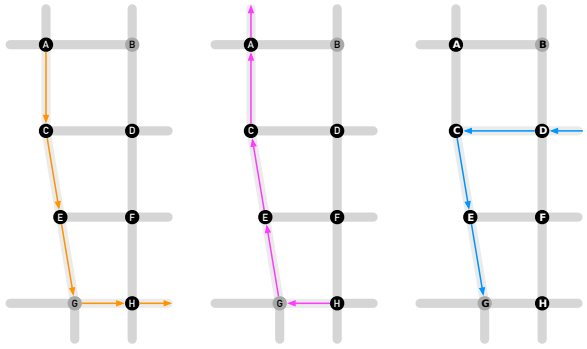
ABSTRACT

With the ever-increasing urbanization, managing traffic and avoiding congestion becomes more and more challenging. Analysing the dynamics of vehicles is a crucial aspect of alleviating the problem of traffic congestion. One key aspect in assessing the traffic situation is to identify *dominant flows*, which are subject to high traffic volumes, and hence, are most prone to generate congestion. Real-world traffic data tracking vehicles in an urban network are proved to be extensive, subject to inconsistencies, and often detections are missing. These render many of the prior techniques inapplicable, highlighting the need for a novel robust and scalable technique. In this paper, we propose IST, an indexing technique for real-life traffic data, a scoring function for identifying the dominant flows and Flow-Scan, an algorithm for querying the dominant flows from the proposed index. Our experimental results demonstrate the efficiency and effectiveness of the presented method. Robustness and effectiveness were tested querying top-k dominant flows on a real-life dataset. In addition, with synthetic data, we demonstrate that our method is scalable while comparing it to related existing methods.

1 INTRODUCTION

Predicting and preventing traffic congestion is important for multiple reasons, such as reducing the driving time from place to place, reducing pollution, reducing waste of fuel, and in general improving the efficiency of urban mobility. In order to prevent traffic congestion in urban roads, it is necessary to analyse the traffic dynamics and identify the areas where traffic jams are more likely to happen. Identifying these potentially problematic areas is the first step to pursue solutions alleviating the issue using traffic optimization techniques.

Dominant flows refer to the longest possible sequences of road segments where a significant number of vehicles travel in a given time window. They are crucial information to enable regional traffic optimization techniques and to coordinate traffic light plans in order to calibrate the capacity of roads. Detecting dominant flows enables smart traffic light plans optimization, allowing on the one hand to increase the throughput of vehicles crossing intersections in the directions affected by the dominant flows (traffic throttling), and on the other hand slow down traffic along with the dominant flows when road capacity cannot be further extended and backpressure must be applied to avoid saturation of road edges, i.e., congestions. Congestions can be interpreted as multiple overlapping flows that constrain each other and so creating traffic jams. In addition, the longer the dominant flows are the more areas they affect. Thus, detecting these flows allows adjusting the traffic light plans to avoid these traffic jams. Once the dominant flows are detected, dedicated analytics can be applied to them for further traffic and movement pattern analysis and for understanding macroscopic traffic trends. Additionally, knowing the dominant flows can be a powerful analytical tool for defining strategies for traffic management and control of traffic authorities.



(a) 03:00-04:00 (b) 08:00-09:00 (c) 13:00-14:00

Figure 1: Example of dominant flows in an area during three different times of a day.

As vehicles are used by people, it is not always straightforward to model their behaviour and moving patterns. For example, Figure 1 presents the dominant flows in an area at three different times of a day. As we can see, the flows are conflicting, especially between the one from 03:00-04:00 that goes from intersection A to intersection H and the one from 08:00-09:00 where it goes from intersection H to intersection A. The most common traffic light systems allow traffic engineers to adjust traffic light plans during the day to accommodate different flows. Thus, existing adaptive traffic light systems seek dominant flow information during the day to avoid traffic jams and shorten commute time.

Our case study concerns an area in one of the largest Chinese metropolis where installed cameras allow to monitor traffic at intersections. Automated video processing allows detecting trajectories of vehicles across intersections, while detecting and tracking their plate number. Summaries of the information extracted from videos are made available for traffic optimization processes at 1 second frequency. The inherent complexity of real-life environment, video processing and software pipelines make the process prone to errors and failures that must be accounted for. This work is part of a solution combining cloud technologies and AI which strives for analysing and coordinating the traffic in the city to alleviate the traffic congestion.

Working with real-life data is a challenging task. Real-life data can be highly incomplete, with respect to missing values, and high noise. Our case study uses data from camera sensors that

*Work done during an internship at Huawei Munich Research Center.

cover only certain parts of the road network, leading to missing information. For example, in Figure 1, there are no camera sensors located in intersection B, so if multiple vehicles traveling from intersection D to intersection B create a dominant flow, this flow will be probably missed. Furthermore, camera sensors might suffer from various problems during their lifetime. Malfunction, bad visibility due to the weather or dirt may prevent precise detection of plate numbers. The aforementioned issues generate various types of inconsistencies in the detection data, and thus making the dominant flow identification more challenging.

The top-k dominant flow extraction is not a new problem. Yet, the previous work [1–3, 6, 9] is considering only the frequency of the flows and not their length. However, if the length of the flows are ignored, single very frequent road segment flows would dominate the top-k results and not allow the longer and more important flows to be identified and analysed. The PSSS method [3] suggests the use of a variant of a fundamental sequential pattern mining method [5, 8]. However, this technique is not applicable in our context due to our challenging real-life data where missing values in the detections are not rare. The method proposed in [3] is using the successor list of each sensor in order to find "hot routes", but in our setting where missing values are common and not all road edges are monitored, the successor lists will not achieve the desired outcome and end up giving wrong results.

This work is focused on identifying dominant flows in given time windows during the day, to support an adaptive traffic light system aimed at optimizing traffic by controlling road capacity and prevent congestion of vehicles on urban roads. In this paper, we aim to discover dominant flows occurring during ad-hoc time windows based on the frequency and length of the flow. The approach we are presenting in this work enables to overcome the issues that the previous methods have with the imperfect real-life data. Furthermore, we consider not only the frequency of a flow, but also its length. The main idea of this work is to use simple data structures to store all the vehicle detections from the camera sensors based both on their timestamps and their location. We adopt a combination of an inverted index and suffix trees to index only the necessary vehicle trip detections. From the index, we are then able to identify the most dominant flows very efficiently in multiple ad-hoc time windows.

Contributions and Outline. In summary, in this work we make the following contributions:

- We introduce a novel scoring function to determine the flows based on a weighted factor between their frequency and length.
- We propose a simple indexing technique for vehicle detections on road networks.
- We propose an efficient algorithm to identify dominant flows on multiple ad-hoc time windows from the proposed index.
- With thorough experiments, we demonstrate that our indexing and querying methods are efficient and viable solutions to our case study.

The rest of the paper is organised as follows. Section 2 reviews the related work, while in Section 3 the problem of finding the top-k dominant flows is introduced. Then, Section 4 describes the indexing function for detecting the dominant flows. We evaluate our approach with the state-of-the-art approaches in Section 5. Finally, Section 6 concludes the paper with the final remarks and provides directions for future work.

2 RELATED WORK

A vehicle trip can be seen as an ordered sequence of detections where each detection represents a single itemset of the sequence. Thus, the identification of dominant flows can be abstracted, and therefore, can be related to the frequent sequential pattern mining topic. One of the early works on this topic is [10] in which the authors made a generalization of a sequential pattern and proposed the GSP algorithm to discover the frequent patterns. Later, two different methods, SPADE [12] and PrefixSpan [5, 8], were proposed that outperformed the GSP method. SPADE [12] was introduced as a method of frequent sequence mining using a lattice structure. PrefixSpan [5, 8] algorithm was later used in other works [3] for various applications like ours. We chose to use this method as our baseline, and not SPADE since this method was used in other works as well, even though none of them perfectly apply in our application, as described in the following Section 5.

An adaptation of PrefixSpan algorithm, the PSSS [3] method was proposed in order to mine hot routes on a road network using private vehicle Electronic Registration Identification data. The problem addressed in [3] can be considered similar to our work and we use the PSSS method as a baseline in our experiments. However, there are two main differences which make PSSS not applicable for our case. First, only the frequency of a route is considered; in our method we use a combination of the frequency and the length of a flow to define it as "hot" or dominant. Second, in PSSS method successor lists for each road node are introduced that determine the neighbouring road nodes where a following detection could happen. These successor lists are used to restrict the search of hot flows only on the neighbouring road edges. Although this makes PSSS method more efficient compared to the PrefixSpan, it cannot be applied in our case. In a real case scenario, we have missing data and gaps in the detections that prevent the successor list of each road node to benefit the identification of hot flows. For example, in Figure 1, the successor list of intersection A includes intersections B and C. If a dominant flow is from intersection A to intersection E through intersection C, and the camera on intersection C is temporarily not giving any detections, this flow would be missed since the search from intersection A will be stopped once there won't be any detections in intersection C. Furthermore, in our case study scenario, there are intersections that do not have detectors and thus by default there are detections missing. The adjacency property of the road network detections is also utilized in the GBM method [6], where it is assumed that the objects are moving in a grid space and each next move is happening to adjacent grid cells. Again, this property is not true for the real case scenarios with incomplete vehicle detection, that, as mentioned before, are due to factor like missing detectors, faulty hardware, poor weather conditions or network failure and the inherent complexity of real-time image processing.

Other works related to the frequent pattern on trajectories include [1, 2, 9] focus on detecting frequent patterns on trajectories that are based on GPS coordinates. These problems are incomparable to ours, with their biggest difference being their use of coordinates floating point data instead of fixed road network point data. Furthermore, only the frequency of the patterns is considered, in contrast to the combination of the length and the frequency that our work adopts.

3 PRELIMINARIES

Let $G = (U, E)$ be a directed unweighted graph that represents a road network. A node $u_i \in U$ represents a road intersection and a directed edge $e_i = (u_x, u_y) \in E$ represents a road segment that starts from intersection u_x and ends in intersection u_y. In each edge of an intersection a detector is located, which gives information as a tuple in the form of (c, t, e), where c is the vehicle id or vehicle plate number, t is the timestamp of the detection and e is the directed road edge where the detection happened. In a real setting where each detector sends data every $1sec$, we can create sequences of detections for each vehicle, defined as vehicle trips.

Definition 3.1 (Trip). A vehicle trip T is an ordered sequence of $o_i = (t_i, e_i)$, where t_i is the timestamp and e_i is the road edge of a detection that the vehicle moved on. To be considered part of the same trip, the time difference between two sequential entries must be less than a manually-defined time threshold t_{min}. Hence, we define a trip as $T = \{o_1, o_2, \ldots, o_{|T|}\}$, where $t_{i+1} - t_i \leq t_{min} \forall i, 1 \leq i \leq |T|$.

A flow $s = \{e_1, e_2, \ldots, e_{|s|}\}$ is an ordered sequence of road edges and its length is defined as the number of road edges that it contains, i.e. $len(s) = |e \in s|$. A trip T contains a flow s at time window $W = [t_a, t_b]$, denoted as $(s, W) \sqsubseteq T$ if and only if the trip T has at least one index i where the sequence of road edges starting at i is the same sequence of ordered road edges during that time window W. More formally $\exists i, 1 \leq i \leq |T| : e_j = o_{i+j-1}.e \; \forall 1 \leq j \leq |s|$ and $[o_i.t, o_{i+|s|}.t] \subseteq W$ and $o_{i+j-1} \in T$ and $e_j \in s$. Given a time window W and a set of trips D_T, we can define the frequency of a flow as the number of trips in which it is contained, i.e. $freq(s, W) = |T_i \sqsubseteq s|$.

In order to avoid considering not useful flows, we use minimum length, l_{min}, and minimum frequency, f_{min}, limits. If a flow has less frequency or less length than the minimum, we ignore it. In addition, we consider only maximal flows, where a flow is considered maximal if there are no sub-flows with the same frequency. A score can get assigned to a flow, based on its frequency and length. In this work the score is the following:

Definition 3.2 (Flow Score). Given a flow s, a time window W, a set of trips D_T, and a parameter $0 \leq \alpha \leq 1$, the score of the flow is the weighted combination of its frequency and length, i.e. $score(s, W, \alpha) = \alpha * freq(s, W) + (1 - \alpha) * len(s)$

Given a number $k \geq 1$, a set of trips D_T, a time window W and a parameter α, the $k-$Flows problem is to identify all the k flows with the highest score.

4 METHOD FLOW-SCAN

The main idea of the proposed method is based on two observations. First, we need to identify the flows, or sub-trips, from the vehicle trips based on their frequency and length. For this purpose, we adopt a structure that utilizes a variant of the suffix tree [11]. Second, we don't need to maintain all the trips from the vehicles, but only the ones that are candidates for the results. In other words, we can discard the trips that have less than the required l_{min}. The trips with less than the required f_{min} need to get retained, since their sub-trips could potentially have more than the required f_{min}.

IST Index. The main data structure of the method is an inverted index. The keys of the inverted index store all the sub-flows with l_{min} that appear in the data. The values of the inverted index are suffix trees with all the sub-flows longer than l_{min} having the key as prefix. The adopted suffix tree, is a compact version of the known suffix tree. Essentially, the nodes that have only one sub-tree are merged with the root of their sub-tree. In this way, the suffix tree has less nodes without loss of the necessary information. In addition, each node of the suffix tree has a pointer to a list of all the trip ids that contain the sub-flow of the node. Using this list, the frequency of the sub-flow can be easily determined.

ID	Path	Vehicle	Timestamps
t1	abcdef	A1	2020/03/20 07:12 – 2020/03/20 07:28
t2	abcg	A2	2020/03/20 07:25 – 2020/03/20 07:41
t3	hdebc	A3	2020/03/20 08:03 – 2020/03/20 08:23

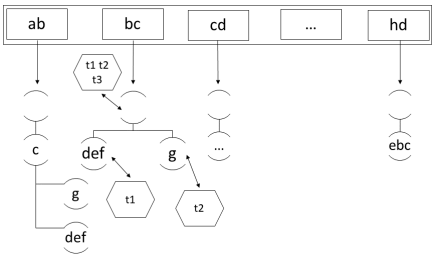


Figure 2: IST index example using $l_{min} = 2$.

An auxiliary data structure is used to store the information of all the trips; *trip id, vehicle id* that made the trip, and the starting and ending timestamps. This data structure is a simple list that is used as supplementary information of the main index. By scanning this list, the trips inside the time window can be identified.

Example 4.1. An example is shown in Figure 2 where the IST index is visualized for three vehicle trips. For this example, we use $l_{min} = 2$. All the sub-flows with length = 2 are indexed as keys in the IST, depicted as rectangles. Each key of the IST index is pointing to a suffix tree, depicted as circles, where all the sub-flows that have the certain keys as prefix are indexed. Furthermore, in the nodes of the suffix trees there are the pointers to the trip ids lists, depicted as hexagons. The IST index is not fully illustrated for presentation reasons. As seen in the figure the sub-flows *bcdef*, *bcg* and *bc* are indexed in the second key of the IST index *bc*. The suffix tree from that key includes the suffixes of the mentioned sub-flows *def* and *g*. The root of the tree is pointing to the list with the trip ids $\{t1, t2, t3\}$, since the sub-flow *bc* is contained in all the trips of our example. Accordingly, the other nodes of the suffix tree contain the respected trip ids in their id lists.

Flow-Scan Query Method. By indexing the trips using the IST data structure, we make sure that no unnecessary trips will become candidates for the dominant flows result and that no information of the trips is lost. In order to query the desired dominant flows, the Flow-Scan process of reading the data from the IST data structure is divided into four steps, shown in Algorithm 1. The first step is the identification of the trips that happen inside the time window W (line 3). This step uses the auxiliary data structure where all the trips are stored is used. Once the trips inside the time window W are retrieved, the keys from the IST inverted index are selected (line 4). This step is important in order to avoid traversing all the data in the IST data structure, but only the elements that could give candidate flows for the result. The third step is to retrieve all the maximal sub-flows that are longer than the l_{min} and more frequent than the f_{min} from

the suffix trees of the keys (lines 5-7). Lastly, the fourth step is to score the retrieved sub-flows using the scoring function from the Definition 3.2 (lines 8-9) and return the k sub-flows with the highest scores.

Algorithm 1 Flow-Scan Algorithm

```
1: function QUERY(IST index, query parameters α, k, W)
2:     Flows ← ∅
3:     D_{T,W} ← FindTripsInWindow(IST, W);
4:     KEYS ← FindIndexKeys(IST, D_{T,W});
5:     for each key ∈ KEYS do
6:         F ← from key.SuffixTree find all maximal flows with
       more than f_min
7:         add all F in Flows
8:     for each s ∈ Flows do
9:         score(s)
       return max_k[Flows]
```

5 EXPERIMENTS

In this section, we report the experimental evaluation of the proposed method, especially for what concerns efficiency and scalability. We first describe the experimental setup, the baseline methods and the datasets used for the evaluation. Then, we describe each experiment, report and discuss the results.

5.1 Environment and Setup

Hardware and Implementation. Our experiments were evaluated on a machine with 24 cores 3.40GHz Intel(R) Core(TM) i7-6700 CPUs, with 16GB memory, using Ubuntu 18.04.4 LTS, and all algorithms are implemented in Java 8.

Compared Algorithms. In our experiments, we use a Naive baseline algorithm to evaluate the efficiency of the Flow-Scan algorithm proposed in this work. The baseline method, Naive, utilizes the structure of the road network to store a list of detections in each road edge e of the network. The list consists of tuples (t, c, p) representing a detection of a vehicle c at time t at the given road edge e. The last, p, is a pointer to the next data point (detection) of the same vehicle c within the same trip T, i.e. the two consequent detections occurred within t_{min}. In this way, the vehicle's trip can be recreated by traversing the vehicle's detections following the pointers p. During the query process, the algorithm scans all the road edges e for detections that occur inside the time window W and it constructs flows by traversing the detections using the pointers p. After identifying all flows, it calculates a score for each flow and returns the k flows with the highest score. The Naive method can handle the missing data from the detections since it uses the road network only to store the detections and not to extract the flows from it, as opposed to the PSSS method [3]. In the Naive method, the search of the flows is happening by following the pointers of the tuples and not following the road network structure.

Additionally with the baseline Naive, we compare the proposed method Flow-Scan with the PrefixSpan method described in [5, 8] and with the PSSS method described in [3]. Since the PrefixSpan and the PSSS methods identify the dominant flows based only on their frequency, we use $\alpha = 1$ for the Naive and Flow-Scan methods in the experiments that all four algorithms are present. Furthermore, since the PrefixSpan and the PSSS methods were proposed for applications that didn't use incomplete data, we use these algorithms only in the experiments with the synthetic datasets as described in the following sections.

Datasets. For the evaluation of the aforementioned methods, we use four different datasets [4], two including real-life data and two synthetic data. The first dataset, RM, is a real-life dataset collected in a period of a month from traffic detectors in an area of a Chinese city. The detectors were located in 6 intersections monitoring the traffic on 44 road edges. Each detector was collecting data every 1 second for all the road edges in its radius. In total, 2894174 detections were collected, from 337089 different vehicles. The second dataset, RD, includes the data from RM for one of the days in that month. The data from that day include 116268 detections from 44643 vehicles.

As already mentioned, these real-life data are incomplete and the PSSS method [3] is not suitable for identifying the dominant flows over them. In order to be able to apply the PSSS and the PrefixSpan [5, 8] algorithms and to experiment with the scalability of the proposed algorithm Flow-Scan we use two datasets, with synthetic data. The synthetic data COM were generated in the same road network as the real-life data RM and RD, with the difference that we included detectors in the 2 intersections where the real scenario was missing. Then we generated equally random trips over the road network. In total we generated 18910 detections from 7500 vehicles. In order to evaluate the scalability of the methods in bigger road networks, we generated the synthetic data GRID using the SUMO Simulation of Urban Mobility [7]. We randomly generated trips on a $10x10$ intersections grid road network (having 100 intersections and 360 road edges) using a utility from SUMO that equally generates trips over the road network. Then, running the simulation and using TraCI Traffic Control Interface library[1] we read the simulation data and collect the detections. The simulation collected data include 1806141 detections from 135618 different vehicles.

5.2 Results and Discussion

Following, the experimental analysis is reported. We show the performance of the methods described based on their building and querying times, and discuss the effect that the various parameters have on the running time of the algorithms. The building time for each algorithm depends on the nature of the algorithm. For all of them the building time includes the reading of the data into memory. For Naive and Flow-Scan it also includes the time for building their respective index. The query time is the time used for each algorithm in order to retrieve the dominant flows from the data or the index. For all algorithms this time includes also the scoring of each flow. For PSSS algorithm the time for creating the successor list *only* for the road edges used in the dominant flows detection is measured in the query time. Note that the experimental results reported are the ones we believe are representative and not the exhaustive set of experiments performed.

Dataset Size Scalability. For testing the scalability, we run the four algorithms, PrefixSpan, PSSS, Naive, Flow-Scan, using the two synthetic datasets COM and GRID for different number of detections and vehicles. All algorithms were run with $t_{min} = 200$ sec, $\alpha = 1, k = 1$, $l_{min} = 2$ and $f_{min} = 2$.

Figure 3 shows the scalability of the building and query times for the four algorithms by changing the number of vehicles in the COM dataset. As expected, the more vehicles there are in

[1] https://sumo.dlr.de/docs/TraCI.html

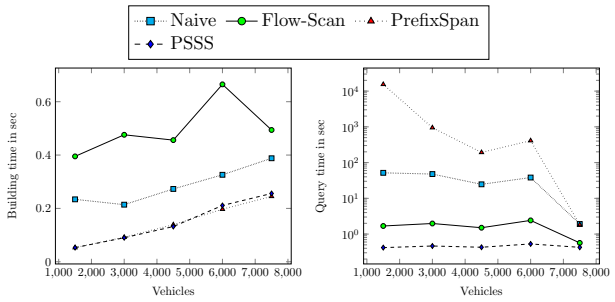


Figure 3: Running times for the COM dataset changing the number of vehicles.

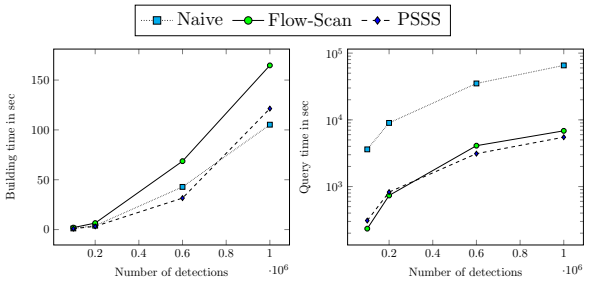
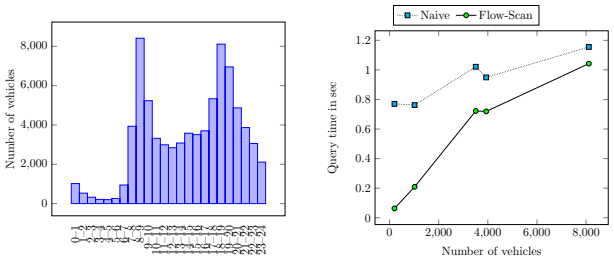


Figure 4: Running times for the GRID dataset changing the number of detections.



(a) Histogram of number of vehicles per hour.

(b) Query time for changing time window W.

Figure 5: Time window analysis for the RD dataset.

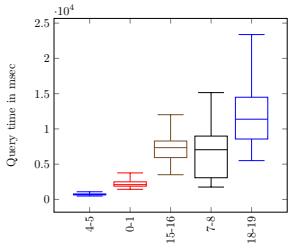


Figure 6: Query time for Flow-Scan method using the RM dataset.

the dataset, the more time the algorithms need for their building step. Method PrefixSpan and PSSS have the same performance on their building phase since they only read the data into memory in this step. On the other hand, the query time does not get affected by the number of the vehicles, excepting the PrefixSpan method that has to iterate multiple times through the vehicle trips to find the candidates for the result.

In Figure 4, the building and query times for the Naive, Flow-Scan and PSSS algorithms are shown by changing the number of detections in the GRID dataset. This dataset was too large for the PrefixSpan method and so we do not report running times for it. Similarly as the COM dataset, with the increase of the number of detections in the data, the building time of the methods is increased. For bigger scale of data, such as the GRID dataset, the building of the IST index is as efficient as the reading of the data into memory that the PSSS method is using, while in the query time the performance is also similar.

Value of Time Window W. The time window W is the query parameter that affects the query time. It is used to restricts the dominant flows in the result, so that they are happening during the time window period. Intuitively, when the time window is in a rush hour period of the day, a significant number of vehicles will travel in the road edges, more flows will become candidates for the result and so, more time will be necessary for the query process. Figure 5a shows a histogram with the number of vehicles per hour for the RD dataset. The peak rush hours are 08:00-09:00 and 18:00-19:00 and the hours with the least traffic are 01:00-06:00.

We run the Naive and the Flow-Scan algorithms for five different time windows that had different number of vehicles. In this experiment we don't use the PrefixSpan and PSSS algorithms since we use the real-life RD dataset that is incomplete. The rest of the parameters were $t_{min} = 200$ sec, $\alpha = 0.5, k = 10, l_{min} = 1$ and $f_{min} = 1$. Figure 5b shows the query times in respect to the number of vehicles that were in each time window. We can see that with more vehicles in the time window, the query time increases. Furthermore, Naive algorithm is affected more by the number of vehicles in the time window, while Flow-Scan algorithm can handle better the increase.

In addition, for validating our method Flow-Scan over one month of data, we run the algorithm for the same time windows for all the days of a month using the RM dataset. The number of vehicles for each hour for all the different days of the month follow the same trend as in Figure 5a. Figure 6 shows the query times for each time window for 30 days. As seen in the figure, the different traffic conditions in the different days do not affect the time windows with small number of vehicles, like the time windows of 04:00-05:00 and 00:00-01:00. As the number of vehicles increases in the time window, the query time differences also increase.

Value of Time Threshold t_{min}. The time threshold affects the indexing time because the higher it is, the more detections will be connected into a single trip. Figure 7 shows the running times for the Naive and Flow-Scan algorithms, run for the RD dataset. Again, the PrefixSpan and PSSS algorithms are not present because of the incomplete real-life dataset. We chose as time window W the period between 07:00-08:00 since this is the period with average number of vehicles. The rest of the parameters were $\alpha = 0.5, k = 10, l_{min} = 1$ and $f_{min} = 1$.

When the time threshold t_{min} is increased, the building time for the Naive index is not affected compared to the building time for the IST index. When the t_{min} is increased, as mentioned, more detections get connected into single trips and the trips grow in length. However, since the vehicle detections stay the same, we get less trips in total, and so the building of the IST index is decreased.

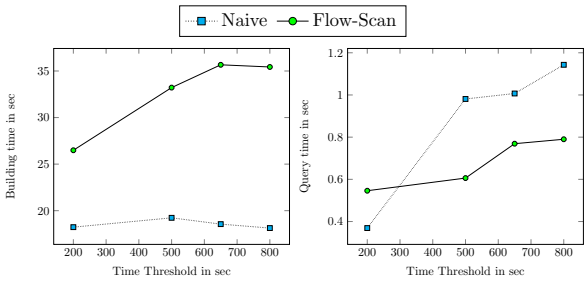


Figure 7: Running times for the RD dataset changing the t_{min} parameter.

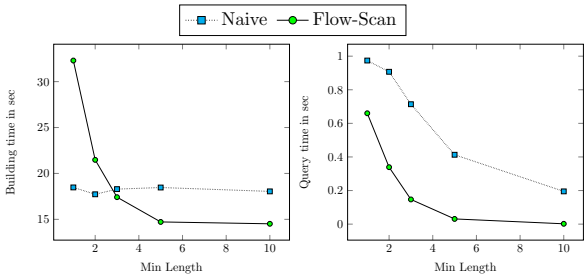


Figure 8: Running times for the RD dataset changing the l_{min} of the trips.

On the contrary, the query time the Naive algorithm gets more affected by the increase of the t_{min} since the trips become longer. The Naive query phase needs to iterate more times in the length of the trips than the Flow-Scan algorithm, and so the increase of the t_{min} increases the query time of Naive.

Value of Minimum Length l_{min}. The l_{min} of the trips controls the length of the flows that can be candidates as dominant flows and thus, indirectly can control how many trips can be considered from the dataset. If a flow is shorter than the l_{min} then it is ignored in the result. In addition, the l_{min} is the value that controls the length of the sub-flows stored as inverted index keys in the IST index. For evaluating how the value of the l_{min} is affecting the Naive and the Flow-Scan methods, we run the algorithms for the RD dataset, with $t_{min} = 500$ sec, $\alpha = 0.5, k = 10$, $f_{min} = 1$ and time window W = 07:00-08:00. As mentioned before, the PrefixSpan and PSSS algorithms are not included in this experiment because of the use of the real-life dataset.

Figure 8 shows the running times for the Naive and the Flow-Scan algorithms. When $l_{min} = 1$, all the flows and subflows are considered as candidates to be dominant, but when the l_{min} increases, the number of flows longer than the threshold decrease, thus reducing the candidate flows. This shows in the query time for both Naive and Flow-Scan algorithms. For both algorithms as the l_{min} is increasing, the query time is decreasing since less flows needs to be analysed. The IST index stores only the trips that have more than the l_{min}, and when the l_{min} is increasing, the index building time is decreasing. The Naive algorithm has more stable building time with the change of the l_{min} and that is because it does not check for l_{min} of the trips before indexing them. However, Naive needs more time for querying than the Flow-Scan, approximately 30% more time, since it keeps all trips and not only the ones longer than l_{min}.

Values of k, α and f_{min}. The parameters k, α and f_{min} affect which of the flows will be detected as the result dominant flows, but do not affect the running time of the algorithms. The parameter k does not affect the algorithms' running times, since the algorithms do not use any pruning or early stopping condition. Similarly, changing α and f_{min} affect only the scoring function of the flows and not the query process of the algorithms.

6 CONCLUSIONS

In this paper, we focused on the real-life scenario of identifying dominant traffic flows, which is crucial for traffic optimization techniques, such as avoiding traffic congestion on urban road networks. We introduced a scoring function that ranks the flows, proposed a simple data structure called IST, and proposed the Flow-Scan algorithm to identify the highest-ranking flows by utilizing the proposed IST index. Our experimental evaluation shows that the Flow-Scan method is efficient and scalable. It is equally efficient on the synthetic datasets with the state-of-the-art PSSS method while at the same time our method enables to overcome the limitations of the PSSS method that does not consider the length of a flow as an important factor and is incapable to handle missing values of the real-life data. Furthermore, although the Flow-Scan method uses more time to build the IST index compared to the Naive method, it outperforms the baseline in the query time and thus, it is a viable solutions for the large scale real-life urban road network scenario studied in this paper.

ACKNOWLEDGMENTS

The authors would like to thank the Lamport team of the Huawei Munich Research Center members for their insightful comments. In addition, they would like to thank Daniele Foroni for his valuable help regarding the datasets and his constructive feedback.

REFERENCES

[1] Huiping Cao, Nikos Mamoulis, and David W Cheung. 2005. Mining frequent spatio-temporal sequential patterns. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*. IEEE, 8–pp.

[2] Lu Chen, Yunjun Gao, Ziquan Fang, Xiaoye Miao, Christian S Jensen, and Chenjuan Guo. 2019. Real-time distributed co-movement pattern detection on streaming trajectories. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1208–1220.

[3] Chen Cui, Linjiang Zheng, and Dihua Sun. 2019. Mining Private Vehicle Hot Routes Using Electronic Registration Identification Data. In *Proceedings of the 2019 International Conference on Big Data Engineering*. ACM, 51–56.

[4] Daniele Foroni, Paolo Sottovia, Stella Maropaki, and Stefano Bortoli. 2021. *Traffic Detection Datasets.* https://doi.org/10.5281/zenodo.4471357

[5] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2001. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*. Citeseer, 215–224.

[6] Anthony J. T. Lee, Yi-An Chen, and Weng-Chong Ip. 2009. Mining Frequent Trajectory Patterns in Spatial-Temporal Databases. *Information Sciences* 179, 13 (2009), 2218–2231.

[7] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie WieBner. 2018. Microscopic traffic simulation using sumo. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2575–2582.

[8] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. 2004. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on knowledge and data engineering* 16, 11 (2004), 1424–1440.

[9] Dimitris Sacharidis, Kostas Patroumpas, Manolis Terrovitis, Verena Kantere, Michalis Potamias, Kyriakos Mouratidis, and Timos Sellis. 2008. On-line discovery of hot motion paths. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. 392–403.

[10] Ramakrishnan Srikant and Rakesh Agrawal. 1996. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology*. Springer, 1–17.

[11] Peter Weiner. 1973. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*. IEEE, 1–11.

[12] Mohammed J Zaki. 2001. SPADE: An efficient algorithm for mining frequent sequences. *Machine learning* 42, 1-2 (2001), 31–60.

Short Paper

On Supporting Scalable Active Learning-based Interactive Data Exploration with Uncertainty Estimation Index

Xiaoyu Ge
University of Pittsburgh
xiaoyu@cs.pitt.edu

Panos K. Chrysanthis
University of Pittsburgh
panos@cs.pitt.edu

ABSTRACT

Driven by the exponential growth in data volume and complexity, and the increasing demand to extract concealed value from it, interactive data exploration (IDE) approaches have recently received a great amount of attention in both industry and academia. To achieve interactiveness, most existing active learning-based IDE systems operate on main-memory databases, which inherently limits the scalability of these IDE systems. In this paper, we propose a novel indexing mechanism, called *Uncertainty Estimation Index* (`UEI`), which supports the interactivity and scalability of the active learning-based IDE systems. `UEI` combines hierarchical in-memory indexing with columnar and inverted-indexing based secondary storage mechanism. It achieves scalability and efficiency through a dynamic estimation of the set of data that are most beneficial to the current exploration. By intelligently manage the in-memory cache, `UEI` enables active learning-based IDE systems to scale beyond the main memory restriction, while maintains the desired accuracy and convergency speed. We experimentally evaluated `UEI` using a state-of-the-art IDE system with two schemes, one incorporating `UEI`, and one utilizing a standard DBMS. We measure the efficiency of the proposed solution using a large real-world dataset placed on the secondary storage. Our experiments show that (1) `UEI` version outperforms the DBMS one by providing more than 50x runtime efficiency when the size of the dataset exceeds the main memory capacity, and (2) is capable of achieving sub-second interactive response time for data that is 100 times larger than the available memory while achieving the desired exploration accuracy and effectiveness.

1 INTRODUCTION

In recent years, as the data has grown rapidly in both complexity and volume, the traditional search methods relying on explicit keywords or queries can quickly lose their effectiveness. As reported in previous studies [15], it is often difficult for users to construct precise articulations that describe their interests. In such cases, traditional search methods usually fail to deliver satisfying results, and the user often needs to deal with results that are too big in size due to loose queries or keywords. Consequently, to obtain a satisfying result, users need to execute numerous ad-hoc queries with tightened conditionals to reduce the search space, which requires a considerable amount of time and human effort. Thus, novel *interactive data exploration* (IDE) techniques that aim to assist users in finding their intended items has generated a significant amount of interest in research communities [2, 7, 9, 10, 12, 13, 16].

One of the core features of these IDE systems is to employ *human-in-the-loop* (HIL) exploration processes to minimize the overall user effort and time in finding the relevant data items. Instead of providing inaccurate results with one generic predictive model (i.e., engine), by leveraging human-in-the-loop, a uniquely learned predictive model can be created rapidly for each individual user and task. To do so, a common solution to support effective human-in-the-loop operations is to leverage active learning techniques [20]—active learning refers to a set of machine learning approaches that aim to learn an accurate predictive model with minimum labeled data for regression and classification tasks. Clearly, the goal of active learning naturally aligns with the needs of many human-in-the-loop techniques as they seek to quickly and accurately deliver the results that the user needs with a personalized, predictive model.

In previous works, numerous active learning techniques [5, 8, 14, 20–22] have been proposed to boost the convergence of training a predictive model. Among these query strategies, *uncertainty sampling* is the most commonly used one because of its simplicity and efficiency, as pointed out in [20]. Uncertainty sampling trains each predictive model in an iterative fashion, wherein each iteration, it identifies the unlabeled items that are closest to the current decision boundary of the predictive model as these items are believed to be most *uncertain*. Uncertainty Sampling then solicits the user's label on the identified sample and utilizes it in the training of the predictive model.

Although uncertainty sampling is more efficient than alternative active learning methods, in order to find the most uncertain object, it still needs to perform an exhaustive search over the entire database. Therefore, in the case where the size of the data is larger than the main memory capacity, those data that resides on the secondary storage must be loaded into memory at each interation. Due to the limitation imposed by physical I/O of the secondary storages, it takes a significant amount of time for active learning techniques to scan datasets that are considerably larger than the main memory. This essentially makes it impossible for any active learning-based IDE system to explore datasets that are larger than the available memory. As pointed in [15], run time efficiency is critical for the IDE systems as any response time excesses 500ms will severely impact the user's engagement, and hence hinders the usability of the system. Moreover, as the exploration could occur on any subset of the attributes of the dataset, it is nearly impossible to apply any typical indexing in advance to support the exploration task over any arbitrary combination of the attributes.

In order to achieve interactiveness, existing uncertainty sampling-based IDE systems rely on main memory to cache the entire dataset. For datasets that are larger than the main memory, a subset of data objects will need to be sampled from the original dataset on secondary storage (e.g., [7]). While simple and intuitive, this approach could easily lead to very inaccurate results and a waste of user effort since the boundaries of the interesting data regions in the sampled space is likely to be different from the original space [9]. Moreover, small sets of relevant data regions may even be ignored in the resulting sample set.

To overcome this problem, we propose a novel indexing mechanism coined *Uncertainty Estimation Index* (UEI), to facilitate the interactivity and scalability of active learning-based IDE systems. To our knowledge, UEI is the first approach in extending the scalability of active learning-based IDE systems beyond the main memory capacity.

Instead of relying solely on the main memory to cache the whole dataset during the exploration, UEI enables caching in memory only the necessary subsets of the data that are needed by the current stage of the exploration. This is achieved through the combination of an effective estimation of the uncertainty of each data object and an efficient data storage mechanism. The essential observation that UEI is based on is that data objects often have additional information that can be used to infer their relationship with other objects, one of which is the *similarity* among data objects. Since the uncertainty essentially represents its distance to the current decision boundary in the high-dimensional data space, thus the uncertainty of an object x is strongly related to the uncertainty of the surrounding objects [14]. This observation allows UEI to informatively select the set of highly uncertain data objects to be loaded into memory before it is needed by the predictive model. Hence, the amount of memory needed to explore a dataset in real-time is significantly reduced.

To evaluate UEI, we employed a state-of-the-art data exploration system REQUEST [9], and compare the performance of UEI against MySQL, which is used by the existing interactive data exploration systems [7, 12]. Our results using a large real-world dataset, namely the Sloan Digital Sky Survey (SDSS) [1], show that UEI outperforms existing solutions by more than 50X in runtime efficiency when the size of the dataset goes beyond the main memory capacity. Furthermore, UEI is well capable of meeting the sub 500 millisecond interactive response time requirement for data that is at least 100 times larger than the main memory capacity, while achieving minimum impact on the convergence of the predictive model.

It should be noted that even though UEI is designed for the IDE systems, where response time and run time efficiency is critical, it can also be used in combination with any active learning-based human-in-the-loop (HIL) applications. Examples of such human-in-the-loop applications including but not limited to: record matching [3], entity resolution [18, 19], and facts checking [4].

2 BACKGROUND

In this section, we provide the necessary background of our `UEI`.

2.1 Active Learning

Active learning refers to a set of approaches that aim to learn an accurate model with minimum labeled data for regression and classification tasks. One key component of active learning is the *query strategy* that sequentially selects the most informative unlabeled sample (i.e., data object) from the entire database to be labeled by the user.

In previous works, a number of active learning techniques have been proposed to define the "informativenes" of samples [20], including: *Uncertainty Sampling* [14], *Query-By-Committee* [21], *Expected Model Change* [5], *Expected Error Reduction* [22], and *Expected Model Output Change* [8]. These techniques are often interchangeable, providing the applications the flexibility to choose the most appropriate query strategy that fits their needs. Among the query strategies, *Uncertainty Sampling* [14] is the most commonly used because of its simplicity and efficiency [20].

Uncertainty Sampling is a query strategy that can be used with any probability-based predictive model (e.g., Naive Bayes, SVM, etc.). The intuition of *Uncertainty Sampling* is that patterns with high uncertainty are hard to classify, and thus, if the labels of those patterns are obtained, they can boost the accuracy of the classification models. Particularly, in binary classification models (e.g., with class labels 0 and 1), the most uncertain example $\mathbf{x}$ is the one which can be assigned to either class label $z(\mathbf{x})$ with probability 0.5. Inspired by the idea of uncertainty, also known as *least confidence* (lc), [14] proposes a measurement of uncertainty for binary classification models:

$$u^{(lc)}(\mathbf{x}) = 1 - p(\hat{y}|\mathbf{x}) \tag{1}$$

where $u^{(lc)}(\mathbf{x})$ is the uncertainty score with the least confidence measurement of $\mathbf{x}$, and $\hat{y}$ is the predicted class label of the unlabeled $\mathbf{x}$. Accordingly, after measuring the uncertainty of each unlabeled sample, the unlabeled sample with highest uncertainty is selected:

$$\mathbf{x}^* = \text{argmax}_{\mathbf{x}} u(\mathbf{x}) \tag{2}$$

where $u(\mathbf{x})$ can be any other measurement of informativeness over the unlabeled sample $\mathbf{x}$.

Algorithm 1 Typical Workflow of Active Learning-based IDE

Require: The raw data set D; Batch Size B
Ensure: A set of results R
1: Labeled set $L \leftarrow \emptyset$
2: Unlabeled set $U \leftarrow D$
3: $M \leftarrow$ initialize query strategy
4: **while** user continues the exploration **do**
5: **for** $i = 1$ to B **do**
6: Choose one x from U using M
7: Solicit user's label on x
8: $L \leftarrow L \cup \{x\}$
9: $U \leftarrow U - \{x\}$
10: **end for**
11: $M \leftarrow$ trained with L to update M.
12: **end while**
13: Return the set of results R classified as positive by M.

2.2 Interactive Data Exploration

Active learning-based IDE systems can effectively find relevant items that are often undiscoverable using traditional search methods [7, 9, 10, 12]. In particular, active learning-based IDE systems do not require users to formulate any complex queries, nor does it need any form of description of the target items. The entire exploration can be done by answering simple binary (i.e., yes or no) questions. More importantly, active learning-based IDE systems can further be used to enhance traditional search results. For instance, they can be used to address the problem of having overwhelming results due to a broad query or search conditions.

As shown in Algorithm 1, a typical active learning-based IDE system works in the following steps: first it incorporates a query strategy (e.g., uncertainty sampling), which is used for selecting the example objects to be presented to the user for labeling (line 3). As long as the user is willing to label more examples (line 4), active learning-based IDE system will keep invoking the query strategy M to select a new example object x from D, and present it to the user to label it as *relevant* or *irrelevant* (lines 5-9). Once the amount of labeled samples received from the user reaches a sample batch size (denoted as B), which is a tunable parameter of the active learning-based IDE balancing the effectiveness and efficiency, then the classifier model employed by the query strategy will be updated according to the label assigned to x (line 11). In particular, the label assigned to x will be used for retraining the classifier model, which is an essential step towards selecting the object presented to the user in the next iteration. Once the iterative labeling process is completed, the obtained results will be presented to the user (line 13).

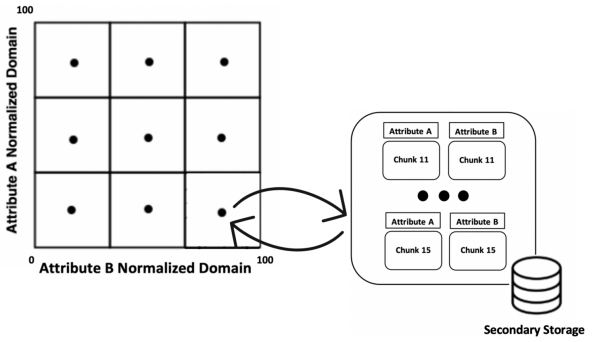


Figure 1: Illustrates UEI with a 2D data space, each grid represents a subspace, the dot in the center of each grid represents a symbolic point p. With chunks stored as separated files on the secondary storage.

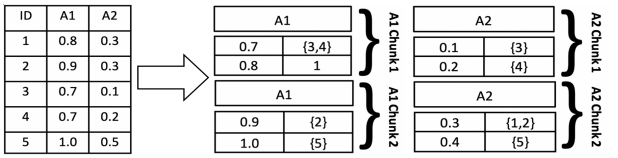


Figure 2: Before storing the data, UEI vertically decompose the data into an inverted index form, and then store them in separate chunks.

3 UNCERTAINTY ESTIMATION INDEX

Maintaining interactive response time for large datasets that are beyond the main memory capacity has always been one of the major challenges of active learning-based IDE systems. In tackling this problem, we focused our efforts on uncertainty sampling and proposed a novel index approach coined *Uncertainty Estimation Index* (UEI). Our UEI can be easily incorporated in any existing systems that leverage the uncertainty sampling and can be used in conjunction with any probabilistic-based classifiers as discussed in [14]. In the next section (Section 3.1) we provide the details of UEI's main component, and discuss how UEI leverages the inverted index-based data secondary storages to support scalable exploration. Following that, in Section 3.2, we provide a walk-through of a complete example to illustrate how a typical active learning-based IDE workflow (i.e., Algorithm 1) performs when enhanced with UEI.

3.1 UEI Components

A key observation underlying UEI is that data objects often have additional information that can be used to inference their relationship with other objects, one of which is the similarity (i.e., distance) between data objects [14]. In other words, the uncertainty value of an object x is strongly related to the uncertainty of the surrounding objects. For example, if x is located near to one of the current decision boundaries, then x would have higher uncertainty due to a mixed set of relevant and irrelevant neighbors. Such continuity between data points is also generally assumed in both supervised and semi-supervised learning algorithms where data points that are closer to each other are more likely to share a label [23]. More precisely, we observed that the uncertainty of a data object x can be approximated through its spatial relationships with the labeled data objects. UEI explores this spatial relationship to load into memory the set of highly uncertain data objects before they are needed by the query strategy.

Specifically, as illustrated in Figure 1, the main idea of UEI is to divide the exploration space D into equal-size subspaces (i.e., d-dimensional grids) g_i's of D ($g_i \in D$), and build a set of symbolic (virtual) index points $P = \{p_1, ...p_c\}$, such that each index point p_i represents a subspace g_i. In each iteration, UEI estimates the uncertainty of each subspace based on the uncertainty of its corresponding index point p, then loads only the data in the subspace that is predicted to be most uncertain into memory. Conceptually, UEI is based on the same principle as hash-based multidimensional indexing, such as the traditional grid file structures, which splits the space into a non-periodic grid where one or more cells of the grid refer to a small set of points. As opposed to a grid file, designed to efficiently reference a single value with multiple keys, UEI is designed to scale out the uncertainty sampling by estimating the distribution of the uncertainty of data objects through the set of symbolic index points.

To do so, UEI comprises five components: 1) an index set P of symbolic index points p_i; 2) a mapping method $m : p \longmapsto C$ that maps each index point p to a set of data chunk C; 3) a data cache U that caches a subset of uniformly sampled unlabeled data; 4) a set of labeled data L that contains all data that has been labeled by the user, and 5) the exploration dataset D stored in a fully inverted columnar format on a hard drive. The first four components reside in the main memory.

UEI divides the operation of exploration into two phases: an *Index Initialization* phase, and *Interactive Exploration* phase. The first phase only needs to be executed once per each new dataset. The second phase is specific to each exploration and discussed in the next section (Section 3.2).

Index Initialization Phase: As illustrated in Algorithm 2, to work with a new dataset, UEI first vertically (i.e., attribute-wise) decompose the whole data set and sort each dimension (i.e., attribute) based on the values in ascending order (lines 2 - 4). Since each dimension of a typical exploratory dataset (e.g., scientific, business analysis) can contain values of an arbitrary length, and one specific value for a dimension may appear multiple times, we compress the data by organizing it in a key-value fashion ($< key, \{values\} >$), where each value of the dimension would be used as a key and the ids of the corresponding objects as $values$ (as illustrated in Figure 2). Note that for each exploration task, UEI stores all needed data in one location, thus when exploring data that are distributed in multiple locations (e.g., tables, files), the data needs to be merged before being utilized in the exploration.

During the process of storing the data, UEI splits the distinct values of each dimension d into a set of equal-sized data chunks $C^d = \{c^d{}_i, ..., c^d{}_u\}$, where each chunk will be stored as a separate file on the disk, and the size of each chunk can be adjusted based on the size of the data and the available hardware resources (line 5). UEI also ensures that the values of each dimension are stored in a sequential order, meaning values stored in each subsequence chunk $c^d{}_{i+1}$ will be larger than the values that have been stored in $c^d{}_i$ for efficient lookup.

Once the data are partitioned and stored on the disk, UEI would start construct the set of symbolic index points P by divide the original data space D into a set of equilateral d-dimensional subspaces $G = \{g_i, ..., g_j\}$, where $|G| = |P|$, then for each subspace g_i, UEI constructs a symbolic index point p_i that represents g_i by using the coordinates of the "virtual" center point of g_i (lines 7-11). To ensure UEI can be deployed in resource restricted environments, the number of symbolic index point can be adjusted based on the size of the dataset and the available hardware resources.

In order to construct and load each subspace g_i into memory, UEI employed a hash-based mapping method m that records for

Algorithm 2 Typical Exploration Flow with UEI

Require: The raw data set D
Ensure: Result set T
1: $P \leftarrow \emptyset, L \leftarrow \emptyset, U \leftarrow \emptyset$
2: $DC \leftarrow verticalDecompose(D)$
3: **for** $d = 1$ to $|DC|$ **do**
4: $sort(DC_d)$
5: $C \leftarrow splitIntoChunks(DC_d)$
6: **end for**
7: $G \leftarrow splitIntoSubspaces(D)$
8: **for** each grid $g_i \in G$ **do**
9: $p_i \leftarrow computeCenter(g_i)$
10: $P \leftarrow P \cup \{p_i\}$
11: **end for**
12: $U \leftarrow sample(D, \gamma)$
13: $M \leftarrow$ initialize predictive model for uncertainty estimation
14: **while** user continues the exploration **do**
15: drop any previously loaded data regions from U
16: $M \leftarrow$ trained with L to update M
17: $P \leftarrow updateUncertainty(P, M)$
18: $p_i^* \leftarrow$ choose the most uncertainty index point from P
19: $g_i^* \leftarrow$ load data region with $m(p_i^*)$
20: $U \leftarrow U \cup g_i$
21: choose one x from U using M
22: solicit user's label on x
23: $L \leftarrow L \cup \{x\}$
24: $U \leftarrow U - \{x\}$
25: **end while**
26: $T \leftarrow resultRetrieval(L)$
27: Return the set of interesting data objects T

each symbolic index point p_i, the set of chunks that are needed to construct g_i. As chunks are stored separately on the disk, this approach allows UEI to quickly identify the data that needs to be loaded. Since each data subspace g is stored as series of one-dimensional data chunks, to reconstruct each g when needed, UEI utilizes a hash table for efficiently merge of those data chunks. During the merge process, UEI iterates through each dimension and loads the corresponding chunks to the memory one at a time, and each entry in the chunk would be visited in a sequential manner. For each object ID that is recorded in a loaded data chunk c_i, the value associated with the ID will be inserted into the corresponding entry in the hash table. Once a chunk has been examined, UEI will release the memory space used to hold the data chunk and reuse the space for the subsequent chunk.

3.2 UEI in Action

In the previous section, we have discussed the components of UEI, as well as how the data are being stored and indexed, which essentially covers the first half (i.e., lines 1 - 11) of the exploration workflow, shown in Algorithm 2. In this section, we will discuss the interactive exploration phase of UEI, which illustrates how a typical active learning-based IDE task (i.e., Algorithm 1) can be performed when incorporating UEI (lines 12 - 27, Algorithm 2).

Interactive Exploration Phase: After the index set has been constructed, UEI begins the exploration by filling the unlabeled set U. Specifically, for the original data space D, UEI would uniformly sample a set of data from the underlying dataset (line 12), where the size of the samples γ can be adjusted based on the system hardware specs (e.g., available main memory size). As a result, a set of unlabeled objects U would be sampled and cached in the main memory. These unlabeled objects will then be used in the acquisition of the set of initial examples that will be labeled by the user to construct the initial predictive model M_0 for uncertainty estimation. Query strategy will randomly sample examples from U until the set of initial examples contains at least one positive example and one negative example (line 13).

In each iteration, UEI updates the uncertainty of all index points $p_i \in P$ based on the most recently trained predictive model M_{t-1} (line 17), which serves as the uncertainty estimator. Here the uncertainty of a data object is essentially equals to the probability of one object being either positive or negative class, with a value that equal to 50% being the most uncertain. Then, the index point p_i^* for which the current exploration model is most uncertain, will be chosen (line 18), such that:

$$p_i^* = \underset{p_i \in P}{\text{Argmax}}\, M_{t-1}(Y|p_i) \tag{3}$$

where Y = {0,1} is set of binary labels.

Based on the chosen p_i^*, UEI uses the mapping method m to identify and load (into the memory) all data chunks that correspond to the subspace g_i^*, which was represented by p_i^* (line 19). As mentioned earlier, the mapping method m is simply a hash table that maps a single index point p into a set of data chunks located on the disk. Later, the data of subspace g_i^* together with the unlabeled dataset U will be used by the query strategy (i.e., uncertainty sampling) in the selection of the example to be labeled in the current iteration (lines 20 - 22). To reduce memory usage, by default UEI kept only one uncertain data region g_i^* in the memory at any given time. Once the user is satisfied with the exploration result, the *resultRetrieval* method will be invoked to retrieve the exploration results and present them to the user (lines 26 - 27).

Tuning Interactive Exploration: In addition to the above typical exploration flow, UEI further allows the user to specify a response latency threshold σ that determines the latency between each exploration iteration (i.e., two subsequent examples). Using the user-specified σ, UEI determines whether or not to defer the swap between the current in-memory uncertain region g_i^* and the next uncertain region g_{i+1}^*, when g_i^* is no longer the most uncertain region.

In the case when an extremely low σ is specified that makes it impossible for the system to load the entire subspace g_i^* into main memory, UEI would start fetching the corresponding data chunks that associated with g_{i+1}^* (in the background) θ iterations before g_{i+1}^* is loaded into the memory. Here, θ is a tunable variable that can also be inferred based on the average loading time τ of data regions, and the configurable latency threshold σ, such that $\theta = \left\lceil \frac{\tau}{\sigma} \right\rceil$.

3.3 Time Complexity of UEI

Clearly, the time complexity of UEI is dominated by the interactive exploration phase. As discussed in Section 3.1, the initialization phase is done once for each dataset and the time required for UEI to prepare and store a dataset D on secondary storage is simply *linear* with respect to the number items n stored in D.

As discussed in Section 3.2, each iteration of the interactive exploration phase in UEI is dominated by the time taken to load the data from the chunks stored on the disk into the memory, which is *linear* with respect to the number of dimensions k and the number entries e stored in the loaded chunks. In contrast, each iteration in the current IDE approaches needs to load and examine all data items n ($e <<< n$).

Therefore, the time complexity of the UEI-enhanced data exploration generally is reduced from $O(kn)$ where n is the number of data objects to $O(ke)$ where e is the number entries associated

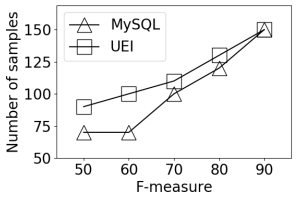


Figure 3: UEI Accuracy (Small Target Region).

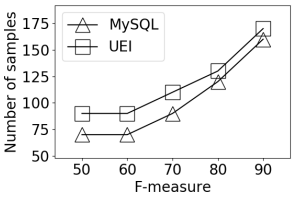


Figure 4: UEI Accuracy (Medium Target Region).

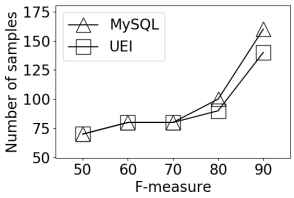


Figure 5: UEI Accuracy (Large Target Region).

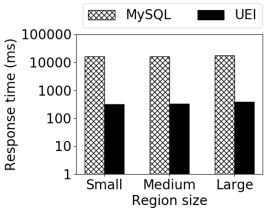


Figure 6: UEI Response Time.

Table 1: PARAMETERS

Number of runs per result	10
Number of dimensions (D)	5
Number of relevant regions	1
Cardinality of relevant regions	0.1% (S), 0.4% (M), 0.8% (L)
Uncertainty Estimator	DWKNN [11]
Label Type	Binary
Data Storage Engine	UEI, MySQL
Size of Individual Data Chunk	470KB
Number of Symbolic Index Points	3125
Latency Threshold	500ms
Performance Measurement	F-Measure (Accuracy)

with the loaded chunks for the current most uncertain subspace g_i^*.

4 EXPERIMENTAL EVALUATION

In our experimental evaluation of our UEI, we use REQUEST [9], a state-of-the-art IDE system, with two schemes, one incorporating UEI, and one utilizing MySQL, used in the existing IDE systems [7, 12]. After describing our experimental setup in Section 4.1, we present the findings of our experimental evaluation in Section 4.2.

4.1 Experiment Setup

Dataset: We used 40 GB of real-world dataset from Sloan Digital Sky Survey (SDSS) [1] that consists of 10×10^6 tuples.

IDE System: In our experiments, we employed our REQUEST [9] with traditional uncertainty sampling and the *dual weighted k-nearest neighbor* (DWKNN) [11] as the uncertainty estimator.

Environment: We implemented both REQUEST and UEI with Java JRE 1.7. All the experiments were run on a machine with Intel Core i7 Processor, 32 GiB RAM and 2 TB of NVMe SSD. The fast NVMe SSD was used to eliminate any potential bottleneck due to physical IO limitation. All experiments reported are averages of 10 complete runs. We have considered five numerical attributes *rowc, colc, ra, dec* and *field* of the PhotoObjAll table.

Target Interest Regions: The exploration task characterizes user interests and eventually predicts the relevant regions by iteratively gathering user labeled tuples. We experimented with 1 region per each exploration task. In addition, we vary the single region complexity based on the data space coverage of the relevant regions. Specifically, we categorize relevant regions to small, medium and large. Small regions have cardinality with an average of 0.1% of the entire experimental dataset, medium regions a cardinality of 0.4%, and large regions a cardinality of 0.8%. Furthermore, the dimensionality of the target interest regions is the same as the dimensionality of the dataset across the entire experiment.

User Simulation For experiment evaluation purpose, we simulate the user behavior using the following method. For each target interest region, we simulate the user by executing the corresponding range query to collect the exact target set of relevant tuples. We rely on this "oracle" set to assign confidence score p to the tuples we extract in each iteration based on their location in the data space against the target region.

More specifically, for each relevant region, there is a region center and a set of region widths, one for each dimension. We define the *maximum relative distance d* of an example against the region center as:

$$d = max_{i=1..l}(|x_i - c_i|/w_i) \quad (4)$$

where l is the dimension number, $|\cdot|$ is the absolute value operator, x_i, c_i and w_i are the attribute value of the example, of the center and region width in each dimension.

Parameters: Table 1 summarizes the important settings in the experiments.

4.2 Experiment Results

In our experiments we aimed to test UEI's ability to provide an interactive response time for datasets that are beyond the size of main memory capacity, and to illustrate the benefit of searching only a small set of cached objects with UEI against performing an exhaustive search over the entire dataset. In our experiments, we stored 10 million data items with both UEI and MySQL, and restricted the memory footprint for both UEI and MySQL to be within 400MB, which is $\sim$ 1% of the entire dataset. Figures 3 to 6 illustrates the effectiveness and efficiency of our proposed UEI.

UEI Accuracy (Figures 3 to 5): Compared to MySQL, we have noticed that our proposed UEI requires more labeled examples in the early stage of the experiment (e.g., below 70% of accuracy). This is due to the fact that in the early stage, the classifier does not have enough training samples to learn an accurate uncertainty estimator (i.e., DWKNN classifier) that captures precisely the user's interesting regions. This causes the predictive model to select less informative examples to be labeled by the user. Since UEI uses uncertainty as the criteria for both the example selection and the loading of data regions (i.e., subspaces), therefore the negative effect of an inaccurate classifier has been magnified.

However, as the accuracy of the uncertainty estimator improves with more labeled examples, we observed a significant boost in performance of UEI in the later stages (e.g., above 80% accuracy) of the exploration. This is expected as the the predictive model gets more and more accurate with respect to the decision boundaries, and thus can estimate more accurate uncertainties. Since UEI rely on the uncertainty estimation for both the query strategy and cache management, therefore, it benefits more noticeably than the MySQL, which only rely on uncertainty estimation for query strategy.

UEI Response time (Figure 6): Finally, we have measured the response time for both UEI and MySQL based schemes. As shown in Figure 6, UEI achieves 50*x* faster response time than MySQL, and ensures the sub-second interactive response time across all data region sizes. Note the response time remains the same across all three target interest regions sizes, which is as expected because the runtime complexity of the uncertainty sampling-based systems only depends on the size of the dataset and not the size of the target interest regions.

From the experiments, it is clear that due to the fact that uncertainty sampling requires an exhaustive search over the entire data spare, thus the physical bandwidth of the secondary storage has become the major bottleneck that severely limits the scalability of active learning-based IDE systems.

Even though in our experiments, we have used NVMe based SSD with I/O throughput of around 3.4GB/s, the uncertainty sampling still takes over 12 seconds to complete the exhaustive search in each iteration. Therefore, it is still impossible to explore datasets that exceed the main memory capacity without UEI.

5 RELATED WORK

Traditionally, indexing has been the core technique for optimizing response time in database systems. Recently, main-memory indexing and specialized access methods have been proposed to support domain-specific query processing and analytics (e.g., [6, 17]). UEI is based on similar principles as these specialized access methods. However, UEI, to the best of our knowledge, is the first domain-specific access method with in-memory and disk components that support interactivity and scalability of active learning-based IDE systems.

The active learning-based IDE systems that can leverage our proposed UEI for better scalability includes *REQUEST* [9], the first active learning-based IDE system, and the two more recently proposed systems, *Dual-Space Model* [12] and *ExNav* [10]. REQUEST utilizes a two stages approach; a data reduction stage aims to selectively reduce the search space while keeping all relevant data regions, and a query selection stage that utilizes an active learning-based predictive model to iteratively improve the accuracy of the constructed exploratory query through interactions with the user. Dual-Space Model uses a new uncertainty sampling-based predictive model and a new dual-space pruning technique that focuses solely on exploration tasks with a single relevant region. It also optimizes these tasks for faster model convergence. ExNav is the first uncertainty sampling-based IDE system that specializes in exploring a variety of unstructured data sets by leveraging the corresponding data embedding methods for each unstructured data type [10].

In addition to the active learning-based IDE systems, UEI can also be utilized in other active learning-based *Human-in-the-loop* (HIL) systems. For example, in [18], the authors have proposed an active learning-based HIL system, called SystemER, for learning Entity Resolution models through user interactions. Another example of an active learning-based HIL application is fact-checking. In [4], the author has proposed an effective active learning-based HIL system for identifying various types of potentially misleading or false information for news contents. Most recently, [19] has proposed to use active learning for learning the implicit structured representations of entity names, which can be useful for many entity-related tasks such as entity normalization and variant generation. To facilitate the process of learning such structured representations, a user-friendly interface called PARTNER has been designed to enhance the user's interaction experience.

6 CONCLUSION

In this paper, we present the *Uncertainty Estimation Index* (UEI), the first indexing mechanism that enables active learning-based IDE systems to explore datasets that exceed the main memory capacity. Instead of requiring all data to be loaded into the main memory as in existing active learning-based IDE systems for sub-second respond times, UEI enables the scalability by dynamically identify and caching the set of objects that are most uncertain to the current stage of the exploration. It achieves this by maintaining a small in-memory index that estimates the aggregated uncertainty value of the data items in the entire data subspaces. UEI also employees columnar-based secondary storage and combines it with an inverted index to support efficient loading of the necessary data items at each iteration.

Our experimental evaluation using real-world data show that a state-of-the-art IDE systems using UEI greatly outperforms a DBMS-based version of the same IDE system by provide more than 50x runtime efficiency when the size of the dataset exceed the main memory capacity, and is capable of achieving sub-second response time for data that is 100 times larger than the available memory while achieving the desired exploration accuracy and effectiveness. In conclusions, UEI can be used not only with existing active learning-based IDE but with other active learning-based Human-in-the-loop systems as well to achieve significantly higher scalability by removing the main memory restriction.

REFERENCES

[1] SDSS Samples Queries - http://cas.sdss.org/dr4/en/help/docs/realquery.asp.
[2] A. Anagnostopoulos, L. Becchetti, C. Castillo, A. Gionis. 2010. An Optimization Framework for Query Recommendation. In *ACM WSDM*. 161–170.
[3] A. Arasu, M. Götz, R. Kaushik. 2010. On active learning of record matching packages. In *ACM SIGMOD*. 783–794.
[4] S. Bhattacharjee, A. Talukder, B. Balantrapu. 2017. Active learning based news veracity detection with feature weighting and deep-shallow fusion. In *IEEE BigData*. 556–565.
[5] W. Cai, Y. Zhang, J. Zhou. 2013. Maximizing Expected Model Change for Active Learning in Regression. In *IEEE ICDE*. 51–60.
[6] G. Chatzigeorgakidis, D. Skoutas, K. Patroumpas, T. Palpanas, S. Athanasiou, S. Skiadopoulos. 2019. Local Similarity Search on Geolocated Time Series Using Hybrid Indexing. In *ACM SIGSPATIAL*. 179–188.
[7] K. Dimitriadou, O. Papaemmanouil, Y. Diao. 2016. AIDE: An Active Learning-Based Approach for Interactive Data Exploration. In *TKDE*, 28:2842–2856.
[8] A. Freytag, E. Rodner, J. Denzler. 2014. Selecting Influential Examples: Active Learning with Expected Model Output Changes. In *ECCV*. 562–577.
[9] X. Ge, Y. Xue, Z. Luo, M. Sharaf, P. Chrysanthis. 2016. REQUEST: A Scalable Framework for Interactive Construction of Exploratory Queries. In *IEEE BigData*. 4566–4579.
[10] X. Ge, X. Zhang, P. Chrysanthis. 2020. ExNav: An Interactive Big Data Exploration Framework for Big Unstructured Data. In *IEEE BigData*.
[11] J. Gou, L. Du, Y. Zhang, T. Xiong, et al. 2012. A new distance-weighted k-nearest neighbor classifier. In *J. Inf. Comput. Sci*, 9(6):1429–1436.
[12] E. Huang, L. Peng, L. Palma, A. Abdelkafi, A. Liu, Y. Diao. 2019. Optimization for active learning-based interactive database exploration. In *VLDB*. 71–84.
[13] S. Islam, C. Liu, R. Zhou. 2013. A framework for query refinement with user feedback. In *J. Syst. Softw., 86(6):1580-1595*.
[14] D. Lewis W. Gale. 1994. A sequential algorithm for training text classifiers. In *ACM SIGIR*. 3–12.
[15] Z. Liu , J. Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. In *IEEE TVCG*, 20(12):2122–2131.
[16] B. McCamish, V. Ghadakchi, A. Termehchy, B. Touri, and L. Huang. 2018. The Data Interaction Game. In *ACM SIGMOD*. 83–98.
[17] B. Peng, P. Fatourou, T. Palpanas. 2020. MESSI: In-Memory Data Series Indexing. In *IEEE ICDE*. 337–348.
[18] K. Qian, L. Popa, P. Sen. 2019. SystemER: A Human-in-the-loop System for Explainable Entity Resolution. *PVLDB* 12, 12, 1794–1797.
[19] K. Qian, P. Raman, Y. Li, L. Popa. 2020. Learning Structured Representations of Entity Names using Active Learning and Weak Supervision. *CoRR* abs/2011.00105.
[20] B. Settles. 2009. *Active learning literature survey*. Technical Report. University of Wisconsin-Madison.
[21] H. S. Seung, M. Opper, and H. Sompolinsky. 1992. Query by Committee. In *ACM Workshop on Computational Learning Theory*.
[22] Y. Zhang, Y. Wang, W. Cai, S. Zhou, Y. Zhang. 2017. From Theory to Practice: Efficient Active Cost-sensitive Classification with Expected Error Reduction. In *SIAM*. 153–161.
[23] Zheng Zhao and Huan Liu. 2007. Semi-supervised Feature Selection via Spectral Analysis. In *SDM*. 641–646.

Short Paper

Efficient Discovery of Approximate Order Dependencies

Reza Karegar
University of Waterloo, CA
mkaregar@uwaterloo.ca

Parke Godfrey
York University, CA
godfrey@yorku.ca

Lukasz Golab
University of Waterloo, CA
lgolab@uwaterloo.ca

Mehdi Kargar
Ryerson University, CA
kargar@ryerson.ca

Divesh Srivastava
AT&T Chief Data Office, US
divesh@att.com

Jaroslaw Szlichta
Ontario Tech Univ, CA
jarek@ontariotechu.ca

ABSTRACT

Order dependencies (ODs) capture relationships between ordered domains of attributes. Approximate ODs (AODs) capture such relationships even when there exist exceptions in the data. During automated discovery of dependencies, *validation* is the process of verifying whether a dependency holds. We present an algorithm for validating AODs with significantly improved runtime performance over existing methods, and prove that it is *minimal* and has *optimal* runtime. By replacing the validation step in a recent algorithm for AOD discovery with ours, we achieve orders-of-magnitude improvements in performance.

1 INTRODUCTION

1.1 Motivation

Functional dependencies (FDs) specify that the values of given attributes *functionally determine* the value of a target attribute. *Order dependencies* extend FDs to state that, additionally, the *order* of tuples with respect to the values from the domains of given attributes determines the *order* of the values from the domain of the target attribute. Table 1 shows a dataset with employee salaries. In this table, the OD that sal *orders* taxGrp holds. If one sorts the table by sal, it is sorted by taxGrp as well.

An OD implies the corresponding FD; e.g., that sal orders taxGrp implies that sal *functionally determines* taxGrp. *Order compatibility* (OC) captures the *co-ordering* aspect of an OD *without* the corresponding FD. Two lists of attributes are *order compatible* if there exists an arrangement for the tuples in the table in which the tuples are sorted according to both. Any OD can thus be equivalently represented by a pair of an OC and an FD [13]. In Table 1, that taxGrp is *order compatible* with sal holds. Note that taxGrp does *not* order sal, as an FD does not hold.

There has been recent work to automate the *discovery* of ODs from data [1, 4, 6, 10, 11]. In practice, however, constraints rarely hold *perfectly* in the data. Real data are dirty, containing wrong and inconsistent values that may violate semantically valid dependencies. This motivates the need for discovering *approximate* ODs (AODs), ODs that hold in the data but with *exceptions*. Discovered ODs deemed semantically valid can be used for data cleaning, to detect erroneous tuples, where measures are then taken to repair the errors [8]. AODs are useful even when the data are not dirty, as there can be exceptions to general rules. AODs help avoid overfitting by discovering more general dependencies.

In Table 1, tax is a fixed percentage of salary in each tax group; i.e., one, three, or eight percent. However, perc includes a concatenated zero in some rows due to data entry errors (e.g., 10% instead of 1% in t_1). Because of this, the OC that salary is order compatible with tax does *not* hold, even though this OC is intended. Similarly, the FD that pos, exp functionally determines sal does *not* hold, due to the exception of tuples t_6 and t_7, two employees with the same position and years of experience but having different salaries. With approximate ODs, we can still discover such concise and meaningful rules in these instances.

Table 1: Employee salaries

#	pos	exp	sal	taxGrp	perc	tax	bonus
t_1	sec	1	20K	A	10%	2K	1K
t_2	sec	3	25K	A	10%	2.5K	1K
t_3	dev	1	30K	A	1%	0.3K	3K
t_4	sec	5	40K	B	30%	12K	2K
t_5	dev	3	50K	B	3%	1.5K	4K
t_6	dev	5	55K	B	30%	16.5K	4K
t_7	dev	5	60K	B	3%	1.8K	4K
t_8	dev	-1	90K	C	8%	7.2K	7K
t_9	dir	8	200K	C	8%	16K	10K

Approximate ODs were introduced in [10]. Their definition of AODs, as is ours herein, is based on the concept of "tuple removal." Given a table and an OD, a *removal set* is a set of tuples which, if removed from the table, results in the OD holding. A *minimal* removal set is one with the smallest cardinality. An *approximation factor* can be defined with respect to a table and an OD, as the ratio of the size of a minimal removal set over the size of the table. For instance, for Table 1 and the OC that pos, exp is order compatible with pos, sal, the minimal removal set and the approximation factor are $\{t_8\}$ and $1/9 \approx 0.11$, respectively.

Given a table $\mathbf{r}$ and an approximation threshold $0 \leq \epsilon \leq 1$, the *discovery problem* for AODs is to find the complete set of minimal *valid* AODs in $\mathbf{r}$ w.r.t. ϵ. Exact ODs are a special case of AODs with an approximation factor of zero. Given a table $\mathbf{r}$, an OD φ, and a threshold ϵ, the problem of *validating* the candidate OD as an AOD involves verifying whether the approximation factor of φ, denoted by $e(\varphi)$, is less than or equal to ϵ.

1.2 Contributions

The extension for AOD discovery in [10, 11], however, is impractical due to its performance. While the approximate FD component can be validated in linear time [3, 10], to validate the approximate OC (AOC) component in the search, they iteratively remove the tuple—or one of the tuples, in the case of a tie—that causes the largest number of violations. This has two weaknesses: the runtime is quadratic in the number of tuples, and it is not guaranteed to find a minimal removal set.

That it is quadratic makes it prohibitively expensive to run on larger datasets. (The validation step for a candidate exact OD has a linear runtime in the number of tuples.) So while the OD discovery algorithm in [10, 11] is shown to scale to datasets with millions of tuples, it is infeasible to run their adapted AOC discovery algorithm over even moderately sized datasets. During

benchmarking, we found in some discovery runs that more than 99% of the running time is spent on validating AOC candidates.

That it deliberately does not guarantee finding a minimal removal set means that the algorithm may overestimate the approximation factor of an AOC candidate. Thus, *true* AOCs with respect to the approximation threshold can be eliminated (while the exact OD discovery algorithm is complete).

In this paper, we resolve this major bottleneck in AOD discovery via an algorithm with optimal runtime and guaranteed minimal removal set for validating AOC candidates. This brings performance of AOD discovery on par with that of OD discovery, while making the AOD discovery complete.

The paper is structured as follows, with the following key contributions. In Sec. 2, we provide background and discuss related work. In Sec. 3.1, we illustrate the established OD and AOD discovery framework—which we then adapt herein—and, in Sec. 3.2, the *iterative validation algorithm* [10, 11] it employs. In Sec. 3.3, we contribute a minimal and optimal validation algorithm based on longest increasing subsequences that decreases the runtime from quadratic to log-linear. In Sec. 4, we present our experimental results, with the following contributions. We demonstrate that AOD discovery using our validation algorithm scales to datasets with millions of tuples and tens of attributes (Exp-1 and Exp-2). We compare our adapted AOC discovery against the previous approach and demonstrate that ours is orders of magnitude faster (Exp-3). As discovering AODs enables the application of pruning rules earlier than for discovering ODs, AOD discovery can be just as efficient, if not more so. Our AOD discovery algorithm gains up to 76% improvement in runtime compared against the (exact) OD discovery algorithm (Exp-5). Given our AOD discovery algorithm is complete, we discover more AODs, and *semantically more general* AODs (thus, of higher quality). We show that we find more AODs, both due to our better scalability and the minimality of our removal sets (Exp-4 and Exp-6). Finally, in Section 5, we conclude with suggestions for future work.

2 PRELIMINARIES AND RELATED WORK

2.1 Definitions and Notation

$\mathbf{R}$ denotes a relational schema, $\mathbf{r}$ represents a table instance, and s and t denote tuples. A and B denote individual attributes and $\mathcal{X}$ and $\mathcal{Y}$ *sets* of attributes. *Lists* of attributes are presented using $\mathbf{X}$ and $\mathbf{Y}$; [] denotes the empty list and $[A \mid \mathbf{T}]$ denotes a list with *head* attribute A and *tail* list $\mathbf{T}$. Tuples t_A and $t_{\mathcal{X}}$ denote the projections of tuple t on A and $\mathcal{X}$, respectively. Wherever a set is expected but a list appears, the list is cast to a set; e.g., $t_{\mathbf{X}}$ is equivalent to $t_{\mathcal{X}}$. $\mathbf{X}'$ represents an arbitrary permutation of the values of a list $\mathbf{X}$ or set $\mathcal{X}$.

Definition 2.1. (nested order) Let $\mathbf{X}$ be a list of attributes where $\mathcal{X} \in \mathbf{R}$. Given two tuples, s and t, $s \preceq_{\mathbf{X}} t$ *iff*

- $\mathbf{X} = [\,]$; *or*
- $\mathbf{X} = [A \mid \mathbf{T}]$ and $s_A < t_A$; or
- $\mathbf{X} = [A \mid \mathbf{T}]$, $s_A = t_A$, and $s \preceq_{\mathbf{T}} t$.

Let $s \prec_{\mathbf{X}} t$ *iff* $s \preceq_{\mathbf{X}} t$ but $t \not\preceq_{\mathbf{X}} s$.

Next, we define order dependencies [1, 4, 6, 10, 11, 13].

Definition 2.2. (order dependency) Let $\mathbf{X}$ and $\mathbf{Y}$ be lists of attributes where $\mathcal{X}, \mathcal{Y} \subseteq \mathbf{R}$. $\mathbf{X} \mapsto \mathbf{Y}$ denotes an *order dependency*, read as $\mathbf{X}$ *orders* $\mathbf{Y}$. Table $\mathbf{r}$ satisfies $\mathbf{X} \mapsto \mathbf{Y}$ ($\mathbf{r} \models \mathbf{X} \mapsto \mathbf{Y}$) iff, for all $s, t \in \mathbf{r}$, $s \preceq_{\mathbf{X}} t$ implies $s \preceq_{\mathbf{Y}} t$. $\mathbf{X}$ and $\mathbf{Y}$ are *order equivalent* (denoted as $\mathbf{X} \leftrightarrow \mathbf{Y}$), iff $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{X}$.

Definition 2.3. (order compatibility) Let $\mathbf{X}$ and $\mathbf{Y}$ be lists of attributes where $\mathcal{X}, \mathcal{Y} \subseteq \mathbf{R}$. $\mathbf{X}$ and $\mathbf{Y}$ are *order compatible*, denoted as $\mathbf{X} \sim \mathbf{Y}$, iff $\mathbf{XY} \leftrightarrow \mathbf{YX}$.

The order dependency $\mathbf{X} \mapsto \mathbf{Y}$ means that $\mathbf{Y}$'s values are monotonically non-decreasing with respect to $\mathbf{X}$'s values. Therefore, if one orders the tuples by $\mathbf{X}$, they are also ordered by $\mathbf{Y}$. The order compatibility (OC) $\mathbf{X} \sim \mathbf{Y}$ means that there exists a total order of the tuples in which they are ordered according to both $\mathbf{X}$ and $\mathbf{Y}$.

Example 2.4. In Table 1, the OD sal $\mapsto$ taxGrp holds. The OC taxGrp $\sim$ sal holds, even though the OD taxGrp $\mapsto$ sal does not.

ODs have a strong correspondence with OCs and FDs. An OD $\mathbf{X} \mapsto \mathbf{Y}$ holds *iff* $\mathbf{X} \sim \mathbf{Y}$ (OC) and $\mathcal{X} \rightarrow \mathcal{Y}$ (FD) hold. This gives two sources of violations for ODs: *swap*s and *split*s [13].

Definition 2.5. (swap) A *swap* with respect to OC $\mathbf{X} \sim \mathbf{Y}$ is a pair of tuples s and t such that $s \prec_{\mathbf{X}} t$ but $t \prec_{\mathbf{Y}} s$.

Definition 2.6. (split) A *split* with respect to FD $\mathcal{X} \rightarrow \mathcal{Y}$ is a pair of tuples s and t such that $s_{\mathcal{X}} = t_{\mathcal{X}}$ but $s_{\mathcal{Y}} \neq t_{\mathcal{Y}}$

Example 2.7. In Table 1, given the OD pos, exp $\mapsto$ pos, sal, tuples t_7 and t_8 constitute a swap (the OC pos, exp $\sim$ pos, sal), and tuples t_6 and t_7 constitute a split (the FD pos, exp $\rightarrow$ pos, sal).

Definition 2.8. tuples s and t are *equivalent* w.r.t. set of attributes $\mathcal{X}$ *iff* $s_{\mathcal{X}} = t_{\mathcal{X}}$. An attribute set $\mathcal{X}$ partitions tuples into *equivalence classes* [3]. The *equivalence class* of tuple $t \in \mathbf{r}$ w.r.t. $\mathcal{X}$ is denoted by $\mathcal{E}(t_{\mathcal{X}})$; i.e., $\mathcal{E}(t_{\mathcal{X}}) = \{s \in \mathbf{r} \mid s_{\mathcal{X}} = t_{\mathcal{X}}\}$. Given a set of attributes $\mathcal{X}$, a *partition* of the table with respect to $\mathcal{X}$ is the set of all equivalence classes; i.e., $\Pi_{\mathcal{X}} = \{\mathcal{E}(t_{\mathcal{X}}) \mid t \in \mathbf{r}\}$.

Example 2.9. In Table 1, $\mathcal{E}(t_{1\{\text{pos}\}}) = \mathcal{E}(t_{2\{\text{pos}\}}) = \mathcal{E}(t_{4\{\text{pos}\}}) = \{t_1, t_2, t_4\}$, and $\Pi_{\text{pos}} = \{\{t_1, t_2, t_4\}, \{t_3, t_5, t_6, t_7, t_8\}, \{t_9\}\}$.

2.2 A Canonical Mapping

A natural representation of ODs relies on lists of attributes, as in the ORDER BY statement in SQL, where the order of attributes in the list matters; e.g., the OD pos, sal $\mapsto$ pos, exp is different than the OD pos, sal $\mapsto$ exp, pos. This is unlike FDs, where the order of attributes does not matter, as with the GROUP BY statement in SQL. Working within this list-based representation, however, has led to discovery frameworks with factorial worst-case runtimes in the number of attributes [6]. Fortunately, lists are *not* inherently necessary to express ODs. In [10, 11], the authors rely on a polynomial mapping of list-based ODs into a logically *equivalent* collection of set-based *canonical* ODs to devise a discovery framework with exponential worst-case runtime in the number of attributes and linear in the number of tuples.

Definition 2.10. (canonical order compatibility) Given a set of attributes $\mathcal{X}$, $\mathbf{X}'A \sim \mathbf{X}'B$ is the OC that states that attributes A and B are *order compatible* within each equivalence class of $\mathcal{X}$. We write this as $\mathcal{X}$: A $\sim$ B in the canonical notation, factoring out the common prefix, and refer to this as a *canonical* OC.

Definition 2.11. (order functional dependency) Given a set of attributes $\mathcal{X}$, the FD that states that an attribute A is *constant* within each equivalence class of $\mathcal{X}$ is equivalent to the list-based OD $\mathbf{X}' \mapsto \mathbf{X}'A$. We write this as $\mathcal{X}$: [] $\mapsto$ A in the canonical notation, and refer to this as an *order functional dependency* (OFD).

Given a canonical OC of $\mathcal{X}$: A $\sim$ B or an OFD of $\mathcal{X}$: [] $\mapsto$ A, the set $\mathcal{X}$ is referred to as the *context* of the respective canonical OC or OFD. Intuitively, the context is the common prefix on the left- and right-side of the corresponding list-based OC or OD.

Canonical OCs and OFDs constitute the canonical ODs; i.e., OD ≡ OC + OFD. The OD of $\mathbf{X}'A \mapsto \mathbf{X}'B$ is logically equivalent to the canonical OC of $\mathcal{X}$: A ∼ B and OFD of $\mathcal{X}$A: [] ↦ B. This is $\mathcal{X}$: A ↦ B written in the canonical form.

Example 2.12. In Table 1, sal and bonus are order compatible w.r.t. the context pos; i.e., {pos}: sal ∼ bonus. In the same table, bonus is constant w.r.t. the context pos, sal; i.e., {pos, sal}: [] ↦ bonus. Therefore, sal orders bonus w.r.t. the context pos; i.e., {pos}: sal ↦ bonus.

This mapping generalizes: an OD $\mathbf{X} \mapsto \mathbf{Y}$ holds *iff* $\mathbf{X} \mapsto \mathbf{XY}$ *and* $\mathbf{X} \sim \mathbf{Y}$. These can be encoded into an equivalent set of canonical OFDs and OCs as follows. In the context of $\mathcal{X}$, all attributes in $\mathcal{Y}$ must be constants. In the context of all prefixes of **X** and of **Y**, the trailing attributes must be order compatible:

$$\mathbf{R} \models \mathbf{X} \mapsto \mathbf{XY} \textit{ iff } \forall A \in \mathbf{Y}.\ \mathbf{R} \models \mathcal{X}\text{: } [] \mapsto A \text{ and}$$

$$\mathbf{R} \models \mathbf{X} \sim \mathbf{Y} \textit{ iff } \forall i, j.\ \mathbf{R} \models [X_1, \ldots, X_{i-1}][Y_1, \ldots, Y_{j-1}]\text{: } X_i \sim Y_j.$$

Thus, list-based ODs can be polynomially *mapped* to a set of *equivalent* canonical ODs; i.e., canonical OCs and OFDs [10, 11]. In this work, we refer to canonical OCs simply as OCs.

Example 2.13. The OD [A, B] ↦ [C, D] is equivalent to the following canonical ODs: {A, B}: [] ↦ C, {A, B}: [] ↦ D, {}: A ∼ C, {A}: B ∼ C, {C}: A ∼ D, and {A, C}: B ∼ D.

While various algorithms have been proposed for discovering ODs, most are not *complete*. The algorithm described in [6] relies on the list-based definition and employs aggressive pruning rules to compensate for its factorial time complexity, but which make it deliberately incomplete. The authors in [4] claim completeness but their algorithm misses ODs in which the same attributes are repeated on the left- and right-hand side. A similar completeness claim has been made in [1], which was shown to be incorrect in [12]. The set-based OD discovery algorithm proposed in [10] does offer a sound and complete discovery of ODs. Thus, we build our algorithm atop the framework introduced in [10].

2.3 Definition of Approximate ODs

We refer to canonical AOCs and approximate OFDs (AOFDs) collectively as AODs. We define AODs based on the fewest tuples that must be removed from a table for an OD to hold. This definition was used for AODs in [10]; their AOC validation step (for the only currently existing AOD discovery algorithm) has a quadratic runtime. For AOFDs, validation takes linear time [3].

Definition 2.14. Given a table **r** and an OD φ, a set of tuples **s** is a *removal set* w.r.t. φ *iff* $\mathbf{r} \setminus \mathbf{s} \models \varphi$. Let $|\mathbf{r}|$ denote the *cardinality* of **r**, the number of tuples in **r**. A removal set **s** is a *minimal* removal set *iff* it has the smallest cardinality over all removal sets; i.e., $|\mathbf{s}| = \min(\{|\mathbf{s}| \mid \mathbf{s} \subseteq \mathbf{r},\ \mathbf{r} \setminus \mathbf{s} \models \varphi\})$. Given **s**, the *approximation factor* $e(\varphi)$ is defined as $|\mathbf{s}|/|\mathbf{r}|$.

Example 2.15. Consider Table 1 and the OC of sal ∼ tax. Here, $\mathbf{s} = \{t_1, t_2, t_4, t_6\}$ and $e(\text{sal} \sim \text{tax}) = 4/9 \approx 0.44$, as $\mathbf{r} \setminus \mathbf{s} = \{t_3, t_5, t_7, t_8, t_9\}$ does not contain any swaps with respect to sal ∼ tax and no smaller set $\mathbf{s}'$ exists such that $\mathbf{r} \setminus \mathbf{s}' \models$ sal ∼ tax.

Given a table **r** and an approximation threshold ϵ, $0 \leq \epsilon \leq 1$, the problem of discovering AODs involves finding all minimal (non-redundant that follow from others) ODs φ such that $e(\varphi) \leq \epsilon$. In this work, we focus on the problem of validating AODs; i.e., verifying whether the approximation factor of a given AOD is less than or equal to a provided threshold. We present an optimal algorithm for doing so and incorporate it into an existing OD discovery framework.

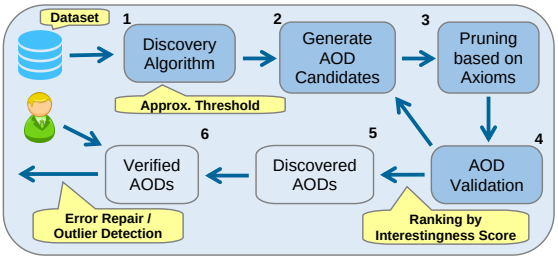


Figure 1: System framework.

As discussed in Sec. 2.2, OCs and OFDs constitute canonical ODs; i.e., OD ≡ OC + OFD. There already exists an efficient linear-time algorithm for validating AOFDs, as described in [3]. In this work, we present an optimal validation algorithm for AOCs. Note that when discovering *approximate* OCs and OFDs given an approximation threshold ϵ, AOD ≡ AOC + AOFD does not necessarily hold. If AOC $\mathcal{X}$: A ∼ B and AOFD $\mathcal{X}$A: [] ↦ B hold with approximation factors $e_1, e_2 \leq \epsilon$, respectively, it is not guaranteed for the corresponding AOD of $\mathcal{X}$: A ↦ B to also hold with respect to ϵ. As to be discussed in Sec. 3.3, however, our validation algorithm can easily be extended to validate list-based approximate ODs as well.

3 DISCOVERING APPROXIMATE OD'S

In Sec. 3.1, we describe our framework to discover set-based canonical AODs. In Sec. 3.2, we describe the iterative validation algorithm proposed in [10, 11], analyze its runtime, and provide an example of it failing to find a minimal removal set and thus overestimating the number of tuples that must be removed. In Sec. 3.3, we present our efficient validation algorithm, based on the longest increasing subsequence (LIS) problem, analyze its runtime, and prove its minimality and optimality.

3.1 Discovery Framework

The algorithm starts the search from singleton sets of attributes and proceeds to traverse the set-based attribute lattice in a level-wise manner [10, 11]. At each level, and when processing the attribute set $\mathcal{X}$, the algorithm verifies AOCs of the form $\mathcal{X} \setminus \{A, B\}$: A ∼ B for which A, B ∈ $\mathcal{X}$ and A ≠ B, and AOFDs of the form $\mathcal{X} \setminus \{A\}$: [] ↦ A for which A ∈ $\mathcal{X}$.

Figure 1 illustrates the framework. Candidate AODs are generated based on the attribute sets at the current level of the lattice. Using the dependencies found in previous levels of the lattice, these candidates are then pruned by axioms to avoid redundant dependencies that follow from already discovered ones [10]. Our algorithm validates whether each candidate dependency holds approximately, given the approximation threshold as input. Valid AODs are then scored and ranked, using the measure of interestingness introduced in [10]. These discovered AODs can then be manually verified by domain experts, to be then used for tasks such as error repair or outlier detection, which is an easier task than manual specification.

3.2 The Iterative Validation Algorithm

We first discuss the algorithm described in [10, 11] to validate an AOC given a threshold ϵ. To validate an AOC, the authors compute a removal set **s** by iteratively removing a tuple with the largest number of swaps, which does not guarantee to produce the minimal removal set. This is repeated until either the OC

Algorithm 1 Approx-OC-iterative

Input: Table $\mathbf{r}$, OC $\mathcal{X}$: A ∼ B, and approximation threshold ϵ.
Output: Approximation factor e and removal set $\mathbf{s}$, or "INVALID"

```
1:  s = {}
2:  for all E ∈ Π_X do
3:      t = order E by [A ASC, B ASC]
4:      t_swapCnt = countInversions(t_B)
5:      order t by swapCnt ASC
6:      while t is not empty do
7:          t = t.dropLast()
8:          if t_swapCnt == 0 then break
9:          for all s ∈ t do
10:             if s_A,B and t_A,B are swapped then s_swapCnt −= 1
11:         end for
12:         order t by swapCnt ASC
13:         add t to s
14:         if |s| > ε|r| then return "INVALID"
15:     end while
16: end for
17: return |s|/|r|, s
```

holds or the number of removed tuples crosses the threshold $\epsilon|\mathbf{r}|$, in which case the AOC candidate is considered invalid. Note that after removing each tuple, the number of swaps for the remaining tuples must be updated.

Algorithm 1 validates a candidate using the iterative approach. The steps in Lines 3 to 15 are repeated on tuples within each equivalence class with respect to the context. Line 4 uses a variant of merge sort to count the number of *inversions* in the projection of sorted tuples over B, which is equivalent to the number of swaps for each tuple. Line 7 removes a tuple with the most swaps and Lines 9 to 11 update the number of swaps for the remaining tuples. Line 14 exits if the approximation threshold is crossed.

Example 3.1. Consider Table 1 and the OC sal ∼ tax. Tuple t_7 has swaps with tuples t_1, t_2, t_4, and t_6, which is more than any tuple in the table, and is thus removed. In following steps, tuples t_5, t_3, t_6, and t_4 are removed. Therefore, $\mathbf{s} = \{t_3, t_4, t_5, t_6, t_7\}$ is reported as a removal set for this AOC, and the approximation factor is computed as $5/9 \approx 0.56$. This is larger than the actual approximation factor for this AOC; i.e., 0.44.

Let m denote the number of tuples in an equivalence class. Lines 3 to 5 have runtime $O(m \log m)$. Lines 7 to 14 inside the loop take $O(m)$ time. Note that since the value of swapCnt for each tuple is bounded by m, sorting the tuples in Line 12 (as well as Line 5) can be done in $O(m)$ time using counting sort. In the worst case, this loop is repeated ϵn times, where ϵ and n denote the approximation threshold and the number of tuples in the table, respectively. Therefore, in the worst case, where $m = n$, the runtime of this algorithm is $O(n \log n + \epsilon n^2)$.

3.3 Our Optimal Validation Algorithm

We now present Algorithm 2 based on the *longest increasing subsequence* (LIS) problem to validate an AOC candidate. Lines 3 to 5 are repeated for the tuples in each equivalence class with respect to the context. Line 3 orders the tuples by [A, B] in ascending order. Next, Line 4 finds a longest non-decreasing subsequence (LNDS) of the projection of tuples over B. (As OCs are symmetric, we can also sort by [B, A] and find a LNDS of projections over A.) Line 5 adds the tuples that are *not* in the LNDS to the removal set. Finally, Line 7 checks whether the OC holds approximately with respect to the threshold, and returns the appropriate output.

Algorithm 2 Approx-OC-optimal

Input: Table $\mathbf{r}$, OC $\mathcal{X}$: A ∼ B, and approximation threshold ϵ.
Output: Approximation factor e and removal set $\mathbf{s}$, or "INVALID"

```
1: s = {}
2: for all E ∈ Π_X do
3:     t = order E by [A ASC, B ASC]
4:     L = computeLNDS(t_B)
5:     s = s ∪ (t_B \ L)
6: end for
7: if |s| ≤ ε|r| then return |s|/|r|, s else return "INVALID"
```

Example 3.2. Consider Table 1 and the OD sal ∼ tax. After ordering the tuples according to sal and breaking ties by tax, the projection of the tuples over tax is the list [2K, 2.5K, 0.3K, 12K, 1.5K, 16.5K, 1.8K, 7.2K, 16K]. The LNDS of this list is [0.3K, 1.5K, 1.8K, 7.2K, 16K] and thus, the removal set is $\mathbf{s} = \{t_1, t_2, t_4, t_6\}$. Thus, the approximation factor is $4/9 \approx 0.44$.

Again, let m denote the number of tuples in an equivalence class. Sorting the tuples in each equivalence class takes $O(m \log m)$ time (Line 3). To compute a LNDS of a list with length m, a dynamic programming algorithm from [2] with small modifications and with runtime $O(m \log m)$ is employed (Line 4). In Line 5, since L is a subsequence of $\mathbf{t}_B$, $\mathbf{t}_B \setminus L$ can be computed in $O(m)$ time by traversing both lists once. Therefore, the worst case runtime of this algorithm, which occurs when $m = n$, is $O(n \log n)$.

We now prove minimality and optimality of our algorithm. [1]

Theorem 3.3. *The set* $\mathbf{s}$ *generated using Algorithm 2 is a minimal removal set with respect to the given* AOC.

Theorem 3.4. *Algorithm 2 has the optimal runtime for validating an* AOC *candidate.*

Our validation algorithm easily extends to AODs of the form $\mathcal{X}$: A ↦ B. We again use Algorithm 2, but in Line 3, tuples are ordered according to the ascending order over A, but ties are broken according to the *descending* order over B. Intuitively, this forces the solution to the LNDS problem in Algorithm 2 to remove all *splits* in the table (removal of *swaps* is already ensured similar to Algorithm 2 for AOCs).[2]

4 EXPERIMENTS

We implemented our approximate OC validation algorithm on top of a Java implementation of the set-based OD discovery framework from [10]. We implemented our new LIS-based algorithm as well as the iterative algorithm using the same technologies to ensure that the improvements in runtime are not due to implementation differences. Unless mentioned otherwise, we set the approximation threshold to 10% and use ten attributes. We run our experiments on a machine with Xeon CPU 2.4GHz with 64GB RAM, and use datasets from the Bureau of Transportation Statistics and the North Carolina State Board of Elections:

(1) **flight** contains information such as date, origin, destination, and airline about flights in the United States and has 1M tuples and 35 attributes (https://www.bts.gov).
(2) **ncvoter** contains information such as registration number, age, and address about voters in North Carolina and has 5M tuples and 30 attributes (https://www.ncsbe.gov).

[1] Due to space limits, proofs of theorems can be found in the technical report [5].
[2] This idea can be extended to list-based AODs of the form $\mathbf{X} \mapsto \mathbf{Y}$, by ordering tuples in ascending order of $\mathbf{X}$ and breaking ties using the descending order over $\mathbf{Y}$.

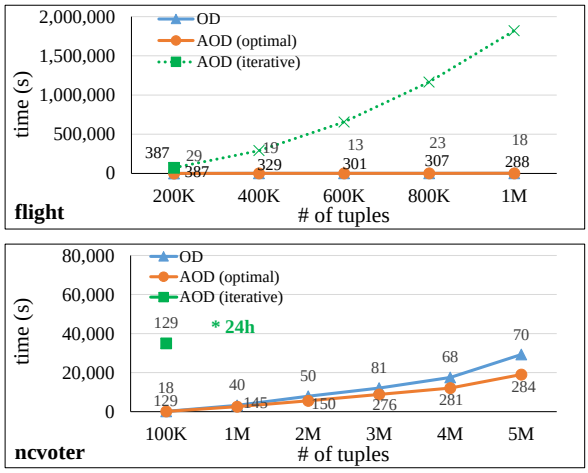


Figure 2: Scalability in |r|.

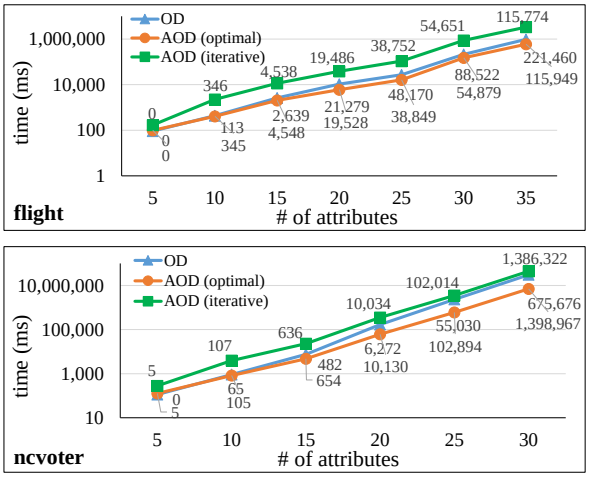


Figure 3: Scalability in |R|.

4.1 Scalability

Exp-1: Scalability in |r|. We measure the runtime (in seconds) of the AOD discovery framework that uses our validation algorithm by varying the number of tuples in our datasets, as reported in Figure 2. For now, ignore the curves labeled "OD" and "AOD (iterative)", as well as the numbers next to the datapoints. The AOD discovery framework implemented using our optimal algorithm scales up to millions of tuples.

Exp-2: Scalability in |R|. Next, we measure the runtime of the discovery framework in milliseconds, by varying the number of attributes in our datasets, as illustrated in Figure 3. We use 1K tuples of our datasets (to allow experiments with a large number of attributes in reasonable time) and vary the number of attributes in multiples of five. In this experiment, the runtime has an exponential growth (the Y-axis in Figure 3 is in log scale). This is expected since the number of ODs increases exponentially with the number of attributes.

4.2 Comparison with the Iterative Algorithm

Exp-3: Runtime comparison with the iterative algorithm. As discussed in Section 3, our AOC validation algorithm has time complexity $O(n \log n)$, while the iterative algorithm proposed in [10, 11] has time complexity $O(n \log n + \epsilon n^2)$. Figures 2, 3, and 4 illustrate the running times of the AOD discovery framework when using these two validation algorithms.

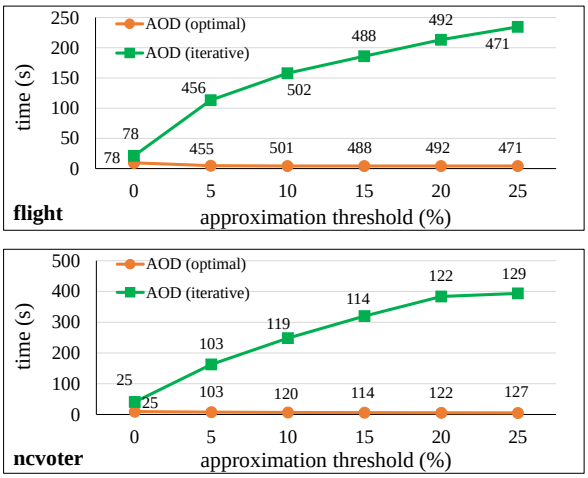


Figure 4: The effect of the approximation threshold.

As shown in Figure 2, while when using our algorithm, the framework can discover AOCs in datasets with up to millions of tuples, when using the iterative algorithm, it does not terminate within 24 hours on 400K and 1M tuples of the flight and ncvoter datasets, respectively (the running times for the flight dataset have been projected with dashed lines for better comparison). In cases where the framework equipped with the iterative algorithm terminates within the time limit, it is orders of magnitude slower. In Figure 3, while the differences are not as pronounced (as the number of tuples is too small), using our validation algorithm still makes the framework almost an order of magnitude faster.

We next experiment with the approximation threshold, by using 10K tuples from our datasets and setting the approximation threshold to 0, 5, 10, 15, 20, and 25 percent. As Figure 4 illustrates, while a larger approximation threshold does not increase the runtime of our algorithm, (the runtime decreases in some cases due to better pruning opportunities), it increases the runtime of the iterative approach at an almost linear rate. This aligns with the time complexity of these algorithms, as analyzed in Section 3.

As mentioned in Section 1, validating AOCs becomes the bottleneck of the AOD discovery framework when using the iterative algorithm. This is verified in our experiments, as up to 99.6% of the total runtime is spent on validation. Using our LIS-based validation algorithm, we reduce the time spent on validating AOCs by up to 99.8%, which results in the orders-of-magnitude improvement in runtime discussed before.

Exp-4: Removal sets and validating AOCs using the iterative algorithm. While our validation algorithm guarantees finding a minimal removal set for a given OC (as is proved in Section 3.3), the iterative algorithm may *overestimate* the size of a minimal removal set. This results in removal sets which are on average around 1% larger than the true minimal removal set.

Overestimating the approximation factor may result in missing valid AOCs if the true approximation factor is close to the input threshold. In Fig. 2, 3, and 4, the numbers inside the plots indicate the number of OCs or AOCs found by an algorithm. We have not listed the number of AOFDs since this work focuses on discovering AOCs. (Wherever the plots for our algorithm and the algorithm for exact ODs overlap, the numbers on the bottom correspond to our approach.) The iterative approach misses up to 2% of the valid AOCs found using our optimal approach.

Missing these AOCs could have potentially severe consequences. For instance, in the flight dataset, the AOC of arrivalDelay ~ lateAircraftDelay holds with an approximation factor of 9.5%.

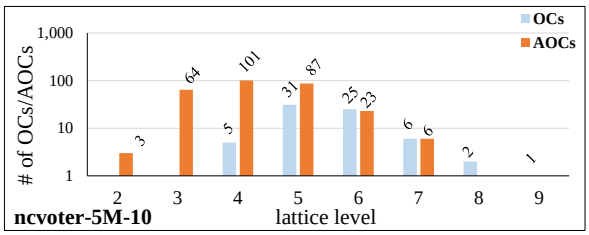


Figure 5: Number of discovered OCs/AOCs in each level.

This AOC points out that generally, delays in arrival are due to the aircraft and not other causes; e.g., security or weather delays. However, the iterative algorithm overestimates the approximation factor as 10.5%. This results in the framework missing this valid AOC when using an approximation threshold of 10%. Note that missing some AOCs results in different pruning opportunities, and, as a result, the set of discovered AOCs, which explains why the iterative algorithm discovers more AOCs in some cases.

Furthermore, as has been discussed for Exp-3, the running time of the iterative algorithm on larger datasets is prohibitively long. On such datasets, using the iterative algorithm results in missing *all* valid AOCs. For instance, in the ncvoter dataset with 5M tuples and with the approximation threshold set to 20%, the AOC of municipalityAbbrv ∼ municipalityDesc is discovered, which points to exceptions in creating abbreviations for municipalities; e.g., "Raleigh" is abbreviated as "RAL", while "Charlotte" is abbreviated as "CLT". However, this AOC does not hold in our 100K sample of tuples when using this threshold. Therefore, this dependency would have been missed by using the iterative validation algorithm, as it exceeds the time limit on the full dataset.

4.3 Comparison with Exact OD Discovery

Exp-5: Lattice level of AOCs and runtime improvements. AOCs tend to reside in lower levels of the lattice (with smaller contexts). In our scalability experiments in the number of tuples (Exp-1), the AOCs are on average 1.2 levels lower on the lattice. Similarly, in experiments in the number of attributes (Exp-2), the AOCs are on average 0.5 levels lower on the lattice. Figure 5 shows the number of OCs or AOCs found at each level of the lattice, when using 5M tuples and 10 attributes of the ncvoter dataset. On this dataset, the average lattice level of the discovered dependencies drops from 5.6 to 4.3 when using our approximate algorithm. As discussed in [10, 11], dependencies found in lower levels of the lattice are likely to be more interesting.

Furthermore, as discussed in Section 3.1, our discovery framework first validates candidates on lower levels of the lattice, and then applies pruning rules to generate the candidates on higher levels of the lattice (step 3 in Figure 1). Therefore, by finding AOCs in lower levels, the algorithm can use pruning rules more effectively earlier in the discovery process, resulting in pruning some candidates on higher levels of the lattice and validating fewer candidates in total. The effects of such pruning opportunities are not noticed when using the iterative validation algorithm, due to its prohibitively long running time. However, we optimally reduce the runtime of the validation step, resulting in runtime improvements for the discovery framework.

Figures 2 and 3 show the running times of the algorithms for discovering exact and approximate ODs. Even though validation of AOCs has a worse runtime compared to exact OCs, i.e., $O(n \log n)$, as opposed to $O(n)$, due to the extra pruning opportunities described above, the total runtime of the discovery framework for AODs can even be lower than the discovery framework for exact ODs; i.e., up to 34% and 76% faster in experiments in the number of tuples and attributes, respectively. The pronounced effect in the experiments in the number of attributes is due to having a smaller number of tuples.

Exp-6: Discovered AOCs compared to OCs. The exact algorithm fails to discover meaningful OCs in presence of anomalies, or even if a single value is erroneous. However, valid AOCs may hold in such instances. Other than the AOCs discussed in Exp-4, in the flight dataset, we discovered the AOC city ∼ airportName with a 27% approximation factor, which indicates that the names of airports usually begin with the name of the corresponding cities. Furthermore, the AOC streetAddress ∼ mailAddress holds in the ncvoter dataset with an approximation factor of 18%. These AOCs can point to anomalies and data quality issues, e.g., wrong address formats, misaligned mailing and residence addresses, and non-standard / erroneous airport names.

As shown in Figures 2 and 3, by discovering AOCs, we can find more dependencies in the data. Even if there are fewer AOCs than OCs (e.g., the flight dataset in Exp-2), the discovered dependencies are on lower levels of the lattice, as shown in Exp-5, which makes them more interesting [10, 11]. If the number of discovered dependencies is too large, the interestingness measure proposed in [11] can be used to rank the AOCs. In fact, the example AOCs that we have identified in Exp-4 and in this experiment, were all ranked as the most interesting AOCs.

5 CONCLUSIONS

We proposed a new validation algorithm for approximate ODs and proved its minimality and runtime optimality. We then implemented our approach in an existing canonical OD discovery framework and demonstrated significant gains compared to existing frameworks for discovering exact and approximate ODs. In future work, we will study new approaches for discovering approximate ODs, such as hybrid sampling, as done in [7] for FDs. We will also extend our approximate OD discovery framework to distributed settings, similar to the work in [9].

REFERENCES

[1] C. Consonni, P. Sottovia, A. Montresor, and Y. Velegrakis. 2019. Discovering order dependencies through order compatibility. *EDBT* (2019), 409–420.
[2] M. Fredman. 1975. On computing the length of longest increasing subsequences. *Discrete Mathematics* 11, 1 (1975), 29 – 35.
[3] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. 1999. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Computer J.* 42 (1999), 100–111.
[4] Yifeng Jin, L. Zhu, and Zijing Tan. 2020. Efficient Bidirectional Order Dependency Discovery. *ICDE* (2020), 61–72.
[5] Reza Karegar, Parke Godfrey, Lukasz Golab, Mehdi Kargar, Divesh Srivastava, and Jaroslaw Szlichta. 2021. Efficient Discovery of Approximate Order Dependencies. *Technical report, 7 pages, http://arxiv.org/abs/2101.02174* (2021).
[6] P. Langer and F. Naumann. 2016. Efficient Order Dependency Detection. *The VLDB Journal* 25, 2 (2016), 223–241.
[7] T. Papenbrock and F. Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. *SIGMOD* (2016), 821–833.
[8] Y. Qiu, Tan, K. Z., Yang, X. Yang, and N. Guo. 2018. Repairing data violations with order dependencies. *DASFAA* (2018), 283–300.
[9] H. Saxena, L. Golab, and I. Ilyas. 2019. Distributed Implementations of Dependency Discovery Algorithms. *PVLDB* 12, 11 (2019), 1624–1636.
[10] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. 2017. Effective and complete discovery of order dependencies via set-based axiomatization. *PVLDB* 10, 7 (2017), 721–732.
[11] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. 2018. Effective and Complete Discovery of Bidirectional Order Dependencies via Set-Based Axioms. *The VLDB Journal* 27, 4 (2018), 573–591.
[12] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. 2020. Erratum for discovering order dependencies through order compatibility. *EDBT* (2020), 659–663.
[13] J. Szlichta, P. Godfrey, and J. Gryz. 2012. Fundamentals of Order Dependencies. *PVLDB* 5, 11 (2012), 1220–1231.

Short Paper

Towards Scalable Data Discovery

Javier Flores, Sergi Nadal, Oscar Romero
Universitat Politècnica de Catalunya
Barcelona, Spain
jflores|snadal|oromero@essi.upc.edu

ABSTRACT

We study the problem of discovering joinable datasets at scale. We approach the problem from a learning perspective relying on profiles. These are succinct representations that capture the underlying characteristics of the schemata and data values of datasets, which can be efficiently extracted in a distributed and parallel fashion. Profiles are then compared, to predict the quality of a join operation among a pair of attributes from different datasets. In contrast to the state-of-the-art, we define a novel notion of join quality that relies on a metric considering both the containment and cardinality proportion between join candidate attributes. We implement our approach in a system called Nextia$_{JD}$, and present experiments to show the predictive performance and computational efficiency of our method. Our experiments show that Nextia$_{JD}$ obtains similar predictive performance to that of hash-based methods, yet we are able to scale-up to larger volumes of data. Also, Nextia$_{JD}$ generates a considerably less amount of false positives, which is a desirable feature at scale.

1 INTRODUCTION

Data discovery requires to identify interesting or relevant datasets that enable informed data analysis [2, 9]. Discovery and integration of datasets is nowadays a largely manual and arduous task that consumes up to 80% of a data scientists' time [19]. This only gets aggravated by the proliferation of large repositories of heterogeneous data, such as *data lakes* [15] or open data-related initiatives [14]. Due to the unprecedented web-scale volumes of heterogeneous data sources, manual data discovery becomes an unfeasible task that calls for automation [11]. Hence, we focus on the very first task of data discovery: the problem of discovering joinable attributes among structured datasets in a data lake. We distinguish three approaches: *comparison by value*, *comparison by hash* and *comparison by profile*. Table 1, overviews recent contributions. Comparison by value relies on auxiliary data structures such as inverted indices or dictionaries to minimize the lookup cost. Alternatively, the comparison by hash approach expects that similar values will collision in the same bucket, also employing index structures for efficient threshold index. Comparison by profile methods leverage on profiles extracted from datasets and their attributes. These are compared to predict whether a given pair of attributes will join.

Search accuracy		
Exact ..... Approximate		
Comp. by value [6, 20, 22]	Comp. by hash [4, 10, 21, 23]	Comp. by profile [5, 7, 8, 12]
Expensive ..... Efficient		
Algorithmic complexity		

Table 1: Overview of approaches by technique, arranged according to accuracy and algorithmic complexity

1.1 Data discovery at scale

Unfortunately, as we experimentally show in Section 3, the state-of-the-art in data discovery does not meet the expectations for web-scale scenarios. Unlike traditional relational databases, these are characterized by *a)* a wide heterogeneity among datasets (e.g., large differences on the number of attributes and / or their cardinalities); *b)* massive volumes of data; and *c)* the presence of a variety of topics, or domains. Overall, these distinguishing features deem current solutions ineffective due to their inability to scale-up as well as the low precision of the results obtained.

Inability to scale-up. Solutions that yield exact results or with a bounded error (i.e., comparison by value and hash) require the construction and maintenance of index structures for efficient lookup. This is a task that becomes highly demanding in terms of computing resources on large-scale datasets. In fact, as we have empirically observed, the available implementations fail to handle datasets of few GBs. Furthermore, most available approaches do not allow incremental maintenance.

Low precision. Comparison by hash solutions employ either containment or Jaccard distance as similarity measures to decide joinability among pairs of attributes. It has been reported, however, that the estimation of such measures is highly imprecise when the cardinality (i.e., the number of distinct values) of an attribute is comparatively larger than the other's [16], which is a common characteristic in real-world web-scale applications. As result, the precision of current approaches is highly affected due to the large number of false positives. To showcase this fact, we designed an experiment collecting 138 datasets from open repositories such as Kaggle and OpenML[1]. Precisely, we devised an heterogeneous collection of datasets ranging different topics, which yielded a total of 110, 378 candidate pairs of string attributes, where 4, 404 of those have a containment higher or

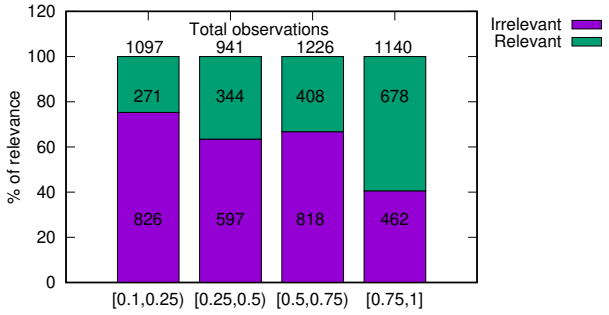


Figure 1: Distribution of relevant and irrelevant results for different containment values on a web-scale repository

[1]Repository available at https://mydisk.cs.upc.edu/s/GeYwdYH7xsGqbaX

equal than 0.1. Indeed, as shown in Figure 1, even for very high containment values (i.e., above 0.75), the number of irrelevant results (i.e., false positives) represents 40% of the total. Such results were obtained by manually analyzing the proposed candidate pairs. However, this approach is unfeasible for large scenarios and better join metrics are needed.

1.2 Profile-based methods to the rescue

The above discussion highlights the limitations of value and hash-based data discovery over web-scale scenarios. Alternatively, the comparison by profile approach suits betters for large scale scenarios as they rely on the detection of similarities or discrepancies between profiles. Working with summaries instead of data values is much more efficient from a complexity point of view. Yet, despite the clear performance benefits of profile-based approaches, there is nowadays a large gap in the trade-off regarding the quality of their results mainly due to the adoption of rather basic profiles (e.g. [8]) that do not accurately describe the underlying data or representative profiles (e.g. [5]) that are used to discover a binary class (e.g. joinable or non-joinable). To that end, we propose a novel approach to data discovery which aims to cover the gap generated by the low predictive performance of profile-based methods, as well as the limited precision and scalability of hash-based systems on large data lakes.

We, first, propose a novel metric to denote the quality of a join. Opposite to the related work, mostly focused on containment or Jaccard distance, we also consider the cardinality proportion between attributes as an indicator of a higher join quality. This allows us to get rid of a substantial amount of false positives, reducing the number of pairs to analyze. This is specially relevant in large-scale settings, where as shown in Figure 1, the number of candidate pairs is too large to manually disregard false positives. Second, we propose a novel learning-based method based on profiles to discover joinable attributes for large-scale data lakes. Our assumptions apply to scenarios where data is typically denormalized and file formats embed tabular data (i.e., not nested). We rely on state-of-the-art relational data profiling techniques [1] to compute informative profiles for datasets. This task, which can be done offline and parallelized over distributed computing frameworks (e.g., Apache Spark), allows us to extract and model the underlying characteristics of attributes. Next, profiles are compared in order to predict their expected join quality. The predictive model is based on random forest classifiers, which are highly expressive and robust to outliers and noise [3]. Additionally, such models can be trained and evaluated in a distributed fashion [17], thus yielding a fully distributed end-to-end framework for data discovery. We show that our method is generalizable and that proposes a meaningful ranking of pairs of attributes based on the predicted join quality.

Contributions. We summarize our contributions as follows:

- We introduce a qualitative metric for join quality, which considers containment and cardinality proportion between attributes.
- We learn a model based on random forest classifiers to efficiently rank candidate pairs of joinable attributes.
- We show that our approach is scalable and outperforms the current state of the art, yielding higher predictive performance results than profile-based solutions and similar quality (F_1-score) to hash-based ones. Yet, our approach yields better precision than hash-based approaches and produce less false positives.

2 MEASURING THE QUALITY OF A JOIN

Unlike the state-of-the-art, which mainly uses containment and Jaccard similarities to decide the degree of joinability among pairs of attributes, we define a qualitative metric to measure the expected join quality. We consider containment as a desirable metric to maximize. Yet, we make the observation that datasets on a data lake do not relate to each other as in a relational database. In such scenarios, it is common to find datasets with few data values in common that, in turn, may represent different semantic concepts. In order to exemplify this idea, let us consider the datasets depicted in Table 2. In this example, the reference dataset D_{ref} might be joined with any of the two candidate datasets D_1 (at the EU level) and D_2 (worldwide). Current approaches would propose both as joinable pairs, since they yield the same containment. However, we aim at distinguishing the join quality between them and use their *cardinality proportion* for that purpose. Let us consider the following cardinalities corresponding to the city attributes: $|D_{ref}|$ = 8124, $|D_1|$ = 54500 and $|D_2|$ = 982921. We use the cardinality proportion as a measure to infer whether their data granularities are similar. In this sense, the third dataset is much larger than $|D_{ref}|$ and yield a worse proportion and therefore we rank it worse. Importantly, we assume these datasets store independently generated events and such big difference in their cardinality most probably mean they embed different semantics or sit at different granularity levels. In general, such situations are a source of false positives for current solutions, specially, when considering small tables.

2.1 Join quality

We now formalize the metric for join quality as a rule-based measure combining both containment and cardinality proportion. We define a totally-ordered set of quality classes S = {`None`, `Poor`, `Moderate`, `Good`, `High`} as indicator of the quality of the resulting join. Indeed, we advocate not to define a binary class (i.e., either joinable or not), since we would not be able to rank the positive ones. As experienced with the state-of-the-art, in realistic large scenarios, a binary class yields a long list of results, which

(a) D_{ref} – Tourism income in Spain

City	Seaside	Amount
Barcelona	Y	350M
Girona	Y	110M
Lleida	N	75M
Tarragona	Y	83M
…	…	…

(b) D_1 – EU demographic data

Unit	Population	Avg. salary	Cost of living
Antwerp	1,120,000	44,000€	2,896€
Barcelona	1,620,343	31,000€	2,422€
Berlin	4,725,000	49,000€	2,737€
Bristol	1,157,937	30,000£	2,397£
…	…	…	…

(c) D_2 – Worldwide demographic data

Name	Country	Population
Barcelona	Spain	1,620,343
Canberra	Australia	426,704
Chicago	United States	2,695,598
Curitiba	Brasil	1,908,359
…	…	…

Table 2: A reference dataset (D_{ref}) and two candidate datasets to be joined. D_1 is curated with extensive data at european level, while D_2 is curated at the worldwide level with less details

are difficult to explore and compare. Therefore, we propose the following multi-class join quality metric.

Definition 2.1. Let A, B be sets of values, respectively the *reference* and *candidate* attributes. The join quality among A and B is defined by the expression

$$Quality(A,B) = \begin{cases} (4)\ \texttt{High}, & C(A,B) \geq C_H \wedge \frac{|A|}{|B|} \geq K_H \\ (3)\ \texttt{Good}, & C(A,B) \geq C_G \wedge \frac{|A|}{|B|} \geq K_G \\ (2)\ \texttt{Moderate}, & C(A,B) \geq C_M \wedge \frac{|A|}{|B|} \geq K_M \\ (1)\ \texttt{Poor}, & C(A,B) \geq C_P \\ (0)\ \texttt{None}, & \text{otherwise} \end{cases}$$

The rationale behind the quality metric is to constrain the candidate pairs of attributes to two thresholds per class: containment (C_i) and cardinality proportion (K_i). Precisely, we fix that for any pair of classes $S_i, S_j \in S$ where $S_i > S_j$, the containment and cardinality proportion must be higher (i.e., $C_H > C_G > C_M > C_P$ and $K_H > K_G > K_M$). Intuitively, a larger containment and a similar cardinality proportion guarantees that the two attributes share common values and their cardinalities are alike. Consequently, most probably, they have a semantic relationship. We consider the values $C_H = 3/4 = 0.75, C_G = 2/4 = 0.5, C_M = 1/4 = 0.25, C_P = 0.1$ for containment, and $K_H = 1/4 = 0.25, K_G = 1/8 = 0.125, K_M = 1/12 = 0.083$ for cardinality proportion. These have been empirically defined from our training set, yet, as we show in Section 3 they are generalizable to other datasets.

To demonstrate the benefits of the proposed metric, we ran an experiment following the same methodology as that depicted in Section 1.1. Using the same collection of 110,378 candidate pairs, we evaluated their join quality. As a result, in Figure 2 we depict the distribution of the join quality distinguishing relevant and irrelevant results. There are two key observations to be made. On the one hand, the number of results labeled with higher quality classes is considerably smaller than those for high containment values. Thus, the largest number of observations are labeled as of *Poor* quality or *None*. On the other hand, the proportion of relevant cases is, in general, much larger than irrelevant ones but specially significative for higher quality classes. As expected, the lower the quality class, the more irrelevant cases will be found. Drilling down in the obtained results, we have manually studied these irrelevant cases where the quality class is High. We have observed that these situations occur when the proposed pairs do not have a semantic relationship but they share a syntactic one (e.g., some artists use a country name). Thus, it would only be possible to disregard them considering semantics.

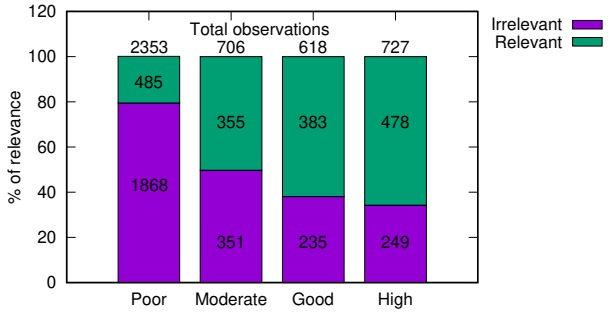


Figure 2: Distribution of relevant and irrelevant results for different quality classes on a web-scale repository

2.2 A learning approach to join discovery

Here, we describe our approach and present the process of building profiles and the predictive model.

Attribute profiling. Profiles are composed of meta-features that represent the underlying characteristics of attributes. Such profiles are the key ingredient for high accuracy predictions, thus we require an exhaustive summary of attributes. To this end, we base our profiling on state-of-the-art relational data profiling techniques [1]. We distinguish meta-features corresponding to unary and binary profiles. We further distinguish the former into meta-features modeling cardinalities, value distribution and syntax. Before comparing profiles and due to the fact attribute meta-features are represented in different magnitudes, we normalize them to guarantee a meaningful comparison using the Z-score. Finally, once meta-features have been normalized we compute the distances among pairs of attributes. Here, we also compute binary meta-features. The result of this stage is a set of distance vectors D where, for each D_i, values closer to 0 denote high similarities.

Predictive model. Once distance vectors are computed, we can train the predictive model. Precisely, the goal is to train a model that, for a pair of attributes A, B, its prediction is highly correlated to the true class (i.e., $Quality(A,B)$). The training process was performed using the collection of datasets discussed in Section 1.1. The ground truth was labeled using the newly proposed quality metric (Definition 2.1), which served as training dataset for the random forest classifiers. In order to reduce the false positive rate, the different classifiers are connected in a classifier chain architecture. This is an effective approach for multi-label classification [18]. Each classifier predicting the probability of class i is trained with the set of distance vectors D and the probabilities of classes $0, \ldots, i-1$, improving the predictive accuracy of the classifier. Figure 3 depicts a high-level overview of the architecture used for training. Then, the prediction returned by the classifier is the one with highest probability from each RF_i. To assign the predicted quality class to a candidate attribute, we assign the label considering the highest probabilities from all classifiers.

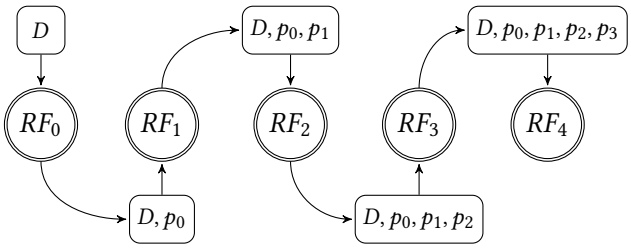


Figure 3: Chain of 5 random forest classifiers, each RF_i is fed with the distances vectors D and the probabilities from the previous classifiers $p_0, \ldots, p_{i-1}$

3 EVALUATION

In this section, we present the evaluation of our approach. On the one hand, we evaluate the ability of the model to discover quality joins through several experiments as well as its generalizability. On the other hand, we compare its performance with representative state-of-the-art solutions. In order to present transparent experiments and guarantee the reproducibility of results, we created an informative companion website[2].

[2]https://www.essi.upc.edu/dtim/nextiajd/

Implementation. Nextia$_{JD}$ is implemented as an extension of Apache Spark. The classification model was trained using its distributed machine learning library MLlib. The runtime methods (i.e., profiling and ranking) are implemented as new operators over the structured data processing library SparkSQL. We leverage on the Catalyst optimizer to efficiently compute the profiles and compare them. Our implementation supports two modes of operation *discovery-by-attribute* and *discovery-by-dataset*. The former receives as input a reference Spark dataframe (i.e., a dataset) and one of its attributes, and generates a ranking against a collection of dataframes (i.e., other datasets). The *discovery-by-dataset* mode does not receive as input a reference attribute, so it runs the discovery process for all attributes from the reference dataframe. Notably, implementing Nextia$_{JD}$ on top of Spark brings many other benefits. Firstly, we can benefit from many source connectors and we can easily ingest the most common data formats (e.g., CSV, JSON, XML, Parquet, Avro, etc.). Secondly, our extension benefits from the inherent capacity of Spark to parallelize tasks on top of distributed data.

Test set. For evaluation purposes, we collected 139 independent datasets from those used for the ground truth. We further divided such datasets into 4 testbeds (extra-small, small, medium and large) according to their file size. Table 3 shows the characteristics of each testbed.

Testbed	***XS***	***S***	***M***	***L***
File size	$0-1$ MB	$1-100$ MB	100 MB -1 GB	>1 GB
Datasets	28	46	46	19
Attributes	159	590	600	331

Table 3: Characteristics per testbed

Alternatives. We compare our approach with the following state-of-the-art data discovery solutions representatives of, respectively, hash-based and profile-based methods whose source code is openly available: LSH Ensemble [23] and FlexMatcher [5]. No fine tuning was performed in such systems, running the code as provided out-of-the-box. These systems differ in their mode of operation, the former being an approach based on comparison by hash, while the latter on comparison by profile. Note that, for computational performance reasons, we rule out approaches based on comparison by value. Such solutions are not comparable to ours, since their kind of search accuracy is exact and by nature they are not suitable for large-scale scenarios.

3.1 Predictive performance

Here we assess the classifier's predictive performance evaluating the ranking of candidate equi-join predicates for each testbed.

Methodology. We depict a confusion matrix to capture the relationship between the true and predicted classes. We also provide performance metrics for the classifier such as precision, recall and F_1 score. We first discuss the experiment for all testbeds together, and later do a fine-grained discussion for each testbed.

Results. Figure 4 and Table 4, show, respectively, the confusion matrix and performance metrics for all testbeds. Overall, we evaluated 467,965 attributes pairs. We can validate the good performance of the proposed approach by the fact that class 4, denoting the highest quality joins, has the best precision. Our method aims at proposing a ranking according to the predicted join quality, which for the highest value it has almost no false positives. It is also relevant to note that the prediction for class 0, denoting the no join quality, also has both high precision and recall. This is particularly relevant to filter out irrelevant results, and thus reduce the search space when presenting a ranking to the user. We additionally note that, as depicted by the precision and recall measures, predictions corresponding to classes 1 and 2 are highly inaccurate. Nevertheless, Nextia$_{JD}$ is prepared to be used as an interactive tool. Thus, if we analyze these results from the point of view of a user, most misclassifications are between similar classes and thus irrelevant. Nextia$_{JD}$ shows the results in strict order. First, classes 4 and 3, and then classes 2 and 1 on demand. In this sense, a binary classification meaning *likely relevant* or *likely irrelevant* would be a fairer way to evaluate Nextia$_{JD}$. When considering these results as a binary problem (relevant: classes 3-4; irrelevant: classes 0-2), the evaluated metrics improve considerably, as shown in Figure 9 and Table 5. Relevantly, we note that Nextia$_{JD}$ generates very few false positives; a desirable property for large-scale data discovery problems. We nevertheless highlight the relevance of distinguishing classes 1 and 2 from 0, either for advanced users or because Nextia$_{JD}$ could be used for other automatic data discovery problems where such distinction would be relevant.

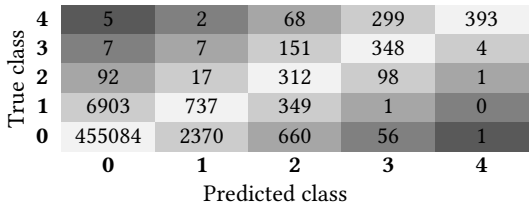


Figure 4: Confusion matrix for all testbeds (clearer cells denote a closer proximity w.r.t. the true class)

	Precision	**Recall**	**F_1 score**
(0) None	0.9848	0.9933	0.9890
(1) Poor	0.2352	0.0922	0.1325
(2) Moderate	0.2025	0.6000	0.3029
(3) Good	0.4339	0.6731	0.5276
(4) High	0.9849	0.5123	0.6740

Table 4: Performance metrics per class for all testbeds

3.2 Comparison with the state-of-the-art

In this experiment we aim at comparing our approach to other data discovery approaches. We perform such evaluation by measuring and comparing their computational complexity and predictive performance.

Methodology. All systems under evaluation, including ours, implement data discovery in two steps. The first step, which we denote *pre*, builds the core data structures from the datasets. For hash-based methods, such as LSH Ensemble, building the index, while profile-based methods, such as FlexMatcher and ours, create the profiles. Additionally, FlexMatcher will create the predictive models for each new data discovery task. Then, the second step, which we denote as *query*, consists of computing the prediction leveraging on the previously built data structures. Whenever possible, we decouple both steps and thus read the data structures from disk. This is, however, not the case for LSH

Ensemble, as it does not offer any resources to store the index on disk, forcing us to maintain it in memory.

The three systems analyzed have slightly different objectives. In order to perform a fair comparison, we analyze the results from the user perspective. That is, the number of results provided and its degree of relevantness. For that, we use a binary scale, which maps to the output obtained in LSH Ensemble and FlexMatcher. For Nextia$_{JD}$, we will reuse the relevantness mapping discussed in the previous experiment: we map classes $\{0, 1, 2\}$ to the irrelevant class, and classes $\{3, 4\}$ to the relevant one. Applying the same rationale, in LSH Ensemble we consider relevant those pairs with a containment above 50% (i.e., the threshold we considered for our class 3). Finally, FlexMatcher is not parameterizable with a quality threshold and already provides a binary output (i.e., non-joinable/joinable). In this case, non-joinable maps to irrelevant and joinable to relevant.

Results. We evaluated the performance of Nextia$_{JD}$, LSH Ensemble and FlexMatcher on each testbed. Both LSH Ensemble and FlexMatcher suffered from scalability issues and were not able to execute testbed *L*. Figure 6, depicts the runtime of the *pre* phase for each testbed. We can observe that the runtime of all systems is in the same orders of magnitude, however our *pre* is larger than the rest. This is mainly due to the fact that, to ensure a fair comparison, we did not set Spark on cluster mode. It is well-known that using Spark on centralized mode adds extra overhead of tasks when generating the required data structures, managing partitions, etc. This is not the case for the other systems, which are provided as standalone programs. Nevertheless, Nextia$_{JD}$ benefits from Spark's robustness and it is the only approach, even in centralized mode, capable of dealing with a large-scale testbed. Furthermore, we note that Nextia$_{JD}$ is the only solution able to precompute its *pre* step (except for binary meta-features).

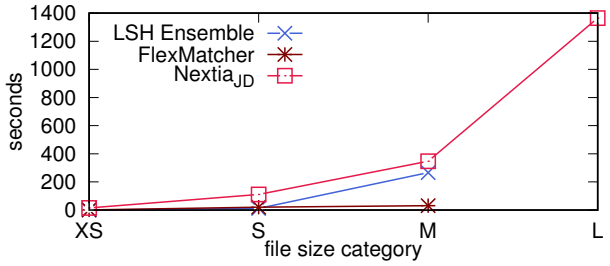


Figure 6: Pre runtime

Regarding the computational performance of the *query* phase, we distinguish the *discovery-by-attribute* and *discovery-by-dataset* scenarios, respectively in Figures 7 and 8. Note that *discovery-by-attribute* is not available in FlexMatcher. LSH Ensemble excels in both *query* tasks. This is due to the fact there is no mechanism to persist the index and this task is reduced to an in-memory lookup. FlexMatcher also benefits from fully running in memory but, in this case, the *query* step suffers from the need to compute some on-the-fly learning models. As general observation, both approaches are thought to compute their core structures and run in memory, which is the main reason hindering their ability to scale-up. Overall, Nextia$_{JD}$ shows a good behaviour in the *query* step. Importantly, Nextia$_{JD}$ is not affected by the dataset cardinality at *query* time. Indeed, the runtime is directly proportional to the number of attributes, or profiles, to compare.

We now put the focus on comparing the predictive performance of the three approaches. Figure 9 and Table 5, depict,

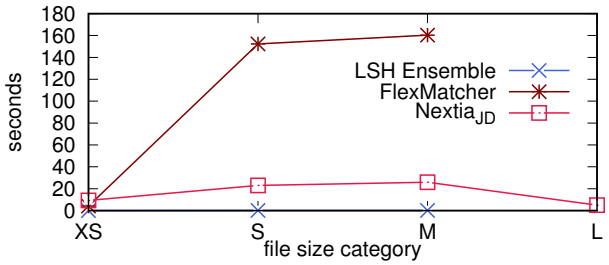


Figure 7: Query runtime (*discovery-by-dataset*)

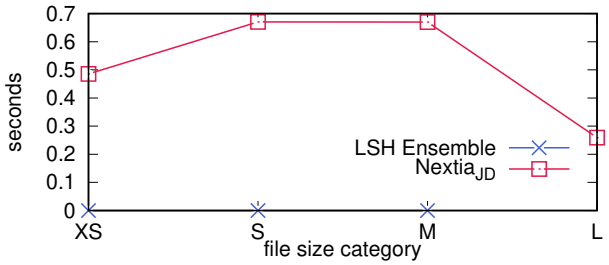


Figure 8: Query runtime (*discovery-by-attribute*)

respectively, the confusion matrices and performance metrics using the binary class mapping. Relevantly, the predictive quality of Nextia$_{JD}$ and LSH Ensemble are comparable. While LSH Ensemble finds more true positives, it generates much more false positives. As result, Nextia$_{JD}$ precision is better. Finally, and aligned with our claim that contemporary profile-based data discovery methods fall short in terms of quality, FlexMatcher generates an extremely large number of false positives reducing its overall quality and making it unfeasible for large-scale scenarios.

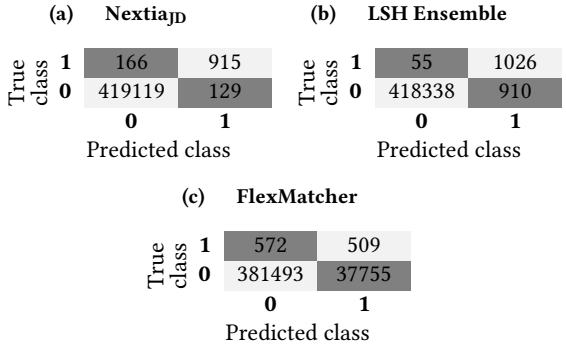


Figure 9: Combined confusion matrices for each system on testbeds *XS*, *S*, *M*

	Precision	Recall	F_1 score
Nextia$_{JD}$	0.8764	0.8464	0.8611
LSH Ensemble	0.5299	0.9491	0.6800
FlexMatcher	0.0133	0.4708	0.0258

Table 5: Performance metrics using binary classes for each system under evaluation on testbeds *XS*, *S*, *M*

Then, Figure 10 drills deeper into the comparison between Nextia$_{JD}$ and LSH Ensemble. We assigned a quality class to LSH Ensemble by running it several times and using a different containment threshold each time, as defined in our quality classes

(i.e., 0.75, 0.5, 0.25 and 0.1). There, we observe relevant differences on the predictions computed. In general, our approach is more conservative, in the sense that we produce less false positives at expenses of sacrificing some true positives. Overall, this improves the precision of our approach by reducing the number of false positives shown to the user. As general observation, both approaches follow slightly different objectives and Nextia$_{JD}$ is more suitable for large-scale scenarios, both for its scale-up capacity and high precision, which guarantees the user will not be overwhelmed with large rankings including false positives.

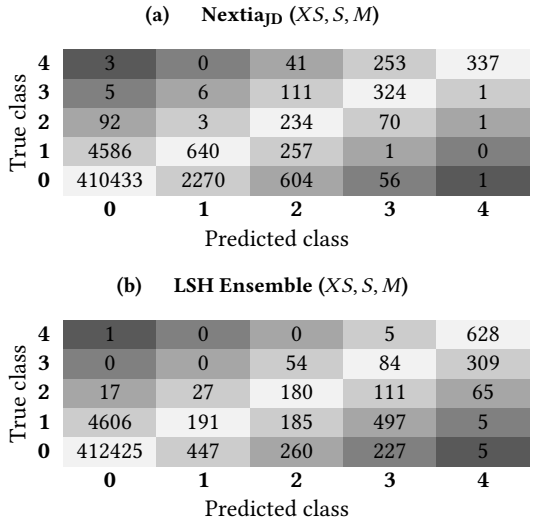


Figure 10: Confusion matrices using 5 quality classes for Nextia$_{JD}$ and LSH Ensemble

4 CONCLUSIONS AND FUTURE WORK

We have presented a novel learning-based approach for data discovery on large-scale repositories of heterogeneous, independently created datasets. Our work is motivated by (i) the poor predictive performance of current profile-based solutions, and (ii) the inability to scale-up and low precision of hash-based ones, which is undesirable for large-scale scenarios. In order to overcome these limitations, we propose a scalable method yielding good precision, and grounded on a novel qualitative definition of join quality. We implemented our approach in a tool called Nextia$_{JD}$. We have experimentally shown that despite being a profile-based approach, Nextia$_{JD}$ presents a similar predictive performance to that of hash-based solutions, yet better adapted for large-scale scenarios, while benefiting from linear scalability.

We do believe profile-based solutions are the right way to go at scale. However, there are some open problems that should be addressed in the future. First, a better join definition metric able to discriminate semantic joins. Current metrics are based on containment / Jaccard similarity and, as previously discussed, these metrics have a very good recall but very low precision. Annotating the required ground truth based on these metrics bias the learning due to the amount of false positives. Current annotated semantic ground truths are unfortunately too small (e.g., Valentine [13]), or generated from approximate metrics like ours, which smoothes the problem but still suffers from it. Last, but not least, profile-based approaches are promising to detect non-syntactic (i.e., with a different encoding per value) semantic join relationships. These are pairs of attributes that maintain the same underlying data distribution but require some transformation in order to join. Based on such predictions, it should be possible to propose such required transformations to join.

ACKNOWLEDGMENTS

This work is partly supported by Barcelona's City Council under grant agreement 20S08704. Javier Flores is supported by contract 2020-DI-027 of the Industrial Doctorate Program of the Government of Catalonia and Consejo Nacional de Ciencia y Tecnología (CONACYT, Mexico).

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *VLDB J.* 24, 4 (2015), 557–581.
[2] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *ICDE*. IEEE, 709–720.
[3] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (2001), 5–32.
[4] Andrei Z. Broder. 1997. On the resemblance and containment of documents. In *SEQUENCES*. IEEE, 21–29.
[5] Chen Chen, Behzad Golshan, Alon Y. Halevy, Wang-Chiew Tan, and AnHai Doan. 2018. BigGorilla: An Open-Source Ecosystem for Data Preparation and Integration. *IEEE Data Eng. Bull.* 41, 2 (2018), 10–22.
[6] Dong Deng, Albert Kim, Samuel Madden, and Michael Stonebraker. 2017. SilkMoth: An Efficient Method for Finding Related Sets with Maximum Matching Constraints. *Proc. VLDB Endow.* 10, 10 (2017), 1082–1093.
[7] AnHai Doan, Pedro M. Domingos, and Alon Y. Halevy. 2003. Learning to Match the Schemas of Data Sources: A Multistrategy Approach. *Mach. Learn.* 50, 3 (2003), 279–301.
[8] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. In *ICDE*. IEEE Computer Society, 1001–1012.
[9] Raul Castro Fernandez, Essam Mansour, Abdulhakim Ali Qahtan, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2018. Seeping Semantics: Linking Datasets Using Word Embeddings for Data Discovery. In *ICDE*. IEEE Computer Society, 989–1000.
[10] Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. 2019. Lazo: A Cardinality-Based Method for Coupled Estimation of Jaccard Similarity and Containment. In *ICDE*. IEEE, 1190–1201.
[11] Behzad Golshan, Alon Y. Halevy, George A. Mihaila, and Wang-Chiew Tan. 2017. Data Integration: After the Teenage Years. In *PODS*. ACM, 101–106.
[12] Mayank Kejriwal and Daniel P. Miranker. 2015. Semi-supervised Instance Matching Using Boosted Classifiers. In *ESWC (Lecture Notes in Computer Science, Vol. 9088)*. Springer, 388–402.
[13] Christos Koutras, George Siachamis, Andra Ionescu, Kyriakos Psarakis, Jerry Brons, Marios Fragkoulis, Christoph Lofi, Angela Bonifati, and Asterios Katsifodimos. 2020. Valentine: Evaluating Matching Techniques for Dataset Discovery. *CoRR* abs/2010.07386 (2020). arXiv:2010.07386 https://arxiv.org/abs/2010.07386
[14] Renée J. Miller, Fatemeh Nargesian, Erkang Zhu, Christina Christodoulakis, Ken Q. Pu, and Periklis Andritsos. 2018. Making Open Data Transparent: Data Discovery on Open Data. *IEEE Data Eng. Bull.* 41, 2 (2018), 59–70.
[15] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *Proc. VLDB Endow.* 12, 12 (2019), 1986–1989.
[16] Azade Nazi, Bolin Ding, Vivek R. Narasayya, and Surajit Chaudhuri. 2018. Efficient Estimation of Inclusion Coefficient using HyperLogLog Sketches. *Proc. VLDB Endow.* 11, 10 (2018), 1097–1109. https://doi.org/10.14778/3231751.3231759
[17] Biswanath Panda, Joshua Herbach, Sugato Basu, and Roberto J. Bayardo. 2009. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. *Proc. VLDB Endow.* 2, 2 (2009), 1426–1437.
[18] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. 2011. Classifier chains for multi-label classification. *Mach. Learn.* 85, 3 (2011), 333–359.
[19] Michael Stonebraker and Ihab F. Ilyas. 2018. Data Integration: The Current Status and the Way Forward. *IEEE Data Eng. Bull.* 41, 2 (2018), 3–9.
[20] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.* 36, 3 (2011), 15:1–15:41.
[21] Yang Yang, Ying Zhang, Wenjie Zhang, and Zengfeng Huang. 2019. GB-KMV: An Augmented KMV Sketch for Approximate Containment Similarity Search. In *ICDE*. IEEE, 458–469.
[22] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD Conference*. ACM, 847–864.
[23] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *Proc. VLDB Endow.* 9, 12 (2016), 1185–1196.

Short Paper

Adaptive Multi-Model Reinforcement Learning for Online Database Tuning

Yaniv Gur
IBM Almaden Research Center
San Jose, CA
guryaniv@us.ibm.com

Dongsheng Yang
Princeton University
Princeton, NJ
dy5@princeton.edu

Frederik Stalschus
DHBW Stuttgart
Stuttgart, Germany
frederik.stalschus@ibm.com

Berthold Reinwald
IBM Almaden Research Center
San Jose, CA
reinwald@us.ibm.com

ABSTRACT

Mainstream DBMSs provide hundreds of knobs for performance tuning. Tuning those knobs requires experienced database administrators (DBA), who are often unavailable for owners of small-scale databases, a common scenario in the era of cloud computing. Therefore, algorithms that can automatically tune the database performance with minimum human guidance is of increasing importance. Developing an automatic database tuner poses a number of challenges that need to be addressed. First, out-of-the-box machine learning solutions cannot be directly applied to this problem and, therefore, need to be modified to perform well on this specific problem. Second, training samples are scarce due to the time it takes to collect each data point and the limited accessibility to query data submitted by the database users. Third, databases are complicated systems with unstable performance, which leads to noisy training data. Furthermore, in a realistic online environment, workloads can change when users run different applications at different times. Although there are several research projects for automatic database tuning, they have not fully addressed this challenge, and they are mainly designed for offline training where the workloads do not change. In this paper, we aim to tackle the challenge of online tuning in evolving workloads environment by proposing a multi-model tuning algorithm that leverages multiple Deep Deterministic Policy Gradient (DDPG) reinforcement learning models trained on varying workloads. To evaluate our approach, we have implemented a system for tuning a PostgreSQL database. The results show that we can automatically tune a PostgreSQL database and improve its performance on OLTP workloads and can adapt to changing workloads using our multi-model approach.

1 INTRODUCTION

Modern DBMSs have hundreds of configuration knobs that affect their performance. A DBMS that is not configured properly for the current workload may lead to sub-optimal performance and inefficient usage of system resources that may result in hundreds of users that are not getting the performance they need for their applications. The role of monitoring and configuring a DBMS was traditionally done by a database administrator (DBA), an expert dedicated to this task. However, nowadays, multiple DMBS instances are deployed on the cloud and each instance could host hundreds of databases, therefore, the task of monitoring and configuring a large-scale database infrastructure requires a large number of DBAs, which would lead to high operation costs.

Over the last few years, several database vendors have identified the potential of using machine learning to automate different database tasks on the cloud, such as automatic indexing, configuration, and provisioning. A few examples include the autonomous database from Oracle [11] and the self-driving database from Alibaba [1]. The study of autonomous databases using AI is a very active research area that already yielded a large number of papers, where the most popular machine-learning paradigm in recent works is reinforcement-learning [7, 9, 14, 18]. Born as a machine-learning branch for solving complex control problems, reinforcement learning is a natural choice the automatic database tuning tasks.

One of the main challenges of operating an automatic DBMS tuning system on the cloud is the fact that the database environment is dynamic: system resources, workloads, and database size could change in the course of the day, therefore, a system for automatic tuning needs to be flexible and adapt to these changes to provide the optimal performance for a given environment state. In this paper, we address the problem of changing workloads in an online tuning setting, and we employ reinforcement learning for this task. While query-aware formulations for tuning were previously proposed [7, 16], the problem of changing workloads in an online tuning setting was not fully addressed.

Our main contributions in this paper are as follows:

- We propose a multi-model online tuning algorithm, sensitive to workload changes, that leverages multiple DDPG reinforcement learning models and selects the optimal model for evolving workloads.
- We propose a simple reward function formulation for offline and online tuning and show that it yields a more stable learning curve compared to previous art [18].
- We demonstrate the offline and online tuning algorithms on a PostgreSQL database and show that the performance of the database can be significantly improved over the baseline default performance.

2 RELATED WORK

In recent years, multiple studies have addressed the problem of automatic DBMS tuning using various machine-learning techniques. In [5] a method called adaptive sampling was used to automate the knob configuration selection by sampling from past experience and in OtterTune [16] Gaussian Process (GP) regression was used to recommend the best knob settings. Reinforcement learning over continuous action configuration space

using the DDPG algorithm was utilized in CDBTune [18]. This method was further extended in QTune [7] where a new algorithm dubbed DS-DDPG was introduced together with a database state change predictor and a query classification algorithm. The approach in this paper also uses DDPG to learn a policy function for the tuning agent, but instead of one model, uses multiple DDPG models for evolving workloads in an online tuning setting.

Reinforcement learning is not limited to database automatic tuning. In several papers, problems such as query optimization [9, 10, 12], index tuning [2, 14] and data partitioning [17] are also solved using this machine learning paradigm.

3 RL FOR DATABASE TUNING

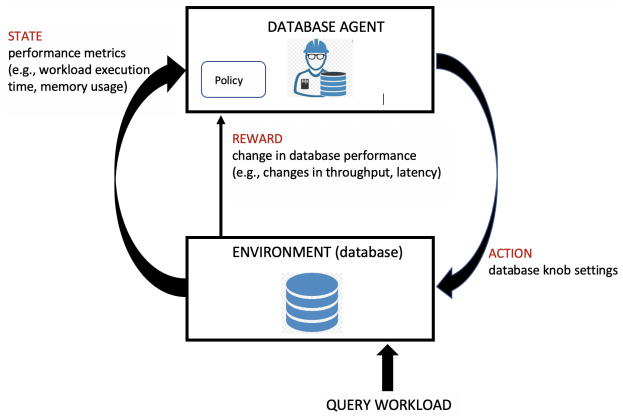


Figure 1: Reinforcement learning applied to database tuning.

A reinforcement learning system is composed of an agent and an environment. The agent takes an action on the environment and the action changes the state of the environment and generates a reward. The state and the reward are fed back into the agent that uses this input to compute the next action. The agent tries to increase the reward in every feedback loop. Reinforcement learning gained its popularity from video games, where the agent simulates a player that takes actions in the game environment and tries to win the game by maximizing a reward function.

As shown in Figure 1, in a database setting, the DBMS represents the environment and the agent represents the DBA. The state at time t, s_t, is described by a set of pre-selected database monitors (statistics collectors) and the reward, r_t, is a function of the database performance metrics (such as throughput and latency). Finally, the action a_t is the change in the configuration knobs selected for automatic tuning. The agent seeks to improve the performance of the database by changing the knob values and sensing the database state and reward.

3.1 DDPG: Knobs tuning in continuous action space

There are many algorithms for reinforcement learning, and each one of them has different variations. One criterion for choosing the right algorithm is whether the action space is discrete or continuous. In discrete action space, the agent selects one action out of a pre-defined set of actions, whereas in the latter case, the agent can pick any action from a continuous action space (e.g., a continuous value). In our case, there are potentially hundreds of knobs to be tuned, and many of them can be tuned to any value in the allowed range of the knob. Therefore, similarly to previous works [7, 18], we selected the Deep Deterministic Policy Gradient (DDPG) RL algorithm, which was developed to tackle the challenge of control problems with a continuous action space [8]. As shown in Figure 2, the agent in the DDPG framework is composed of actor and critic neural networks. Based on the state and reward observations from the environment, the critic estimates a Q-value using the Bellman equation and target networks, and the actor computes the tuning action based on the estimated Q-value. In every tuning iteration, the actor and critic networks are updated n times using batches of samples from a replay buffer that stores the (s_t, a_t, r_t, s_{t+1}) experience tuples. The batch size and the number of update iterations are hyperparameters determined by the user of the algorithm.

Here we describe the selections we made for the key elements of the algorithm:

Reward function. To train the agent, one must define a meaningful reward function to model the database performance. Different reward function formulations had been proposed in previous works [7, 18]. These formulations take into account the initial throughput and latency and the throughput and latency in the previous iteration. In this work, we rely on these formulations, but we use a simpler function that only represents a change in the performance relative to an initial throughput and latency. Our reward function is defined as follows:

$$r_{T,L} = c_T \left(\frac{T}{T_0} - 1 \right) + c_L \left(\frac{L_0}{L} - 1 \right) \quad (1)$$

where T and L stands for throughput and latency, respectively, and $C_T + C_L = 1$. Improvement in the throughput and latency yields positive reward values, whereas negative values indicate a decline in the database performance. T_0 and L_0 are set up as baseline performance values, for example, the throughput and latency obtained by the DBMS default configuration. Due to the typical oscillatory behavior of an RL training process, omitting the dependency on previous iterations adds stability to the tuning process. As will be shown in the experiments section, this formulation together with the selection of $\gamma = 0$ (discount factor) results in a more stable tuning curve compared to [18].

Replay buffer. In a traditional replay buffer, the order in which the experience tuples are saved in the buffer has no importance, and the batch of samples is selected randomly. In this work, we use a prioritized replay buffer in which tuples are prioritized based on the agent's training loss error, therefore, samples are selected based on their importance, and the ones that are more valuable for the learning process would be selected more frequently. It was shown in [8] that prioritized replay buffers lead to a more efficient learning process.

Finally, we followed the idea proposed in [8] and added Gaussian action space exploration noise using the *Ornstein-Uhlenbeck process* [15].

3.2 Offline training

In offline training, the RL agent is trained using databases built specifically for this task and the workloads that are being submitted are configured by the user. The offline training process allows us to generate preliminary models and knob settings for specific workloads, and to tune the algorithm hyperparameters in a controlled environment.

The workload in offline training can be generated and submitted using a benchmarking tool for example, such as Sysbench

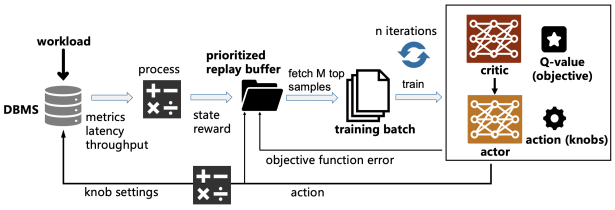


Figure 2: Database tuning with model training using the DDPG reinforcement learning algorithm.

and OLTPBench [4, 6], and the throughput and latency measures can be retrieved from the output log files generated by the tool. Once a workload is chosen, it will not be modified during the training phase. When the training phase is completed, a trained DDPG model for the workload is generated and this model can be used as a starting point for training on a different workload (transfer learning), or as a preliminary model for online tuning, a process we will describe in the next section.

The process of offline training is described in Figure 2. A submitted workload and a change in the knob settings by the action a_t cause a change in the database state and yield s_{t+1}. The throughput and latency measured over a certain time interval obtain r_t using the reward function defined in Eq. 1. The state s_{t+1} and r_t form a state transition tuple with the action a_t and the previous state s_t, and this transition tuple is added to the replay buffer. Then, a batch from the replay buffer is sampled and used to update the actor and critic networks for n iterations. Finally, a new knob settings action a_{t+1} is computed. As part of this process, the batch sample priorities are updated based on the critic loss function. The training process continues for a number of iterations that is determined by the time it takes the learning curve to stabilize.

Offline training was used in previous works for building preliminary machine learning tuning models [7, 16, 18].

3.3 Model deployment

When a trained model is deployed, the algorithm described in Figure 4 is used to recommend the knob settings. When the agent receives the database state and performance metrics, the actor predicts an action that is being translated to knob settings after a Gaussian noise is generated using the OU process and added to the action. These knob settings are then applied to the database.

4 ADAPTIVE MULTI-MODEL ALGORITHM FOR ONLINE TUNING

Tuning online is a challenging task, since workloads can change at anytime. Suppose that a database runs OLAP queries at night and OLTP queries during the daytime, its optimal memory allocation strategy would be different for each workload. Therefore, the training data from an OLAP workload cannot be used for training a model for an OLTP workload, and if the tuning system cannot detect workload changes and adapt, it would perform poorly on new workloads.

To deal with the challenge of changing workloads, we have developed a system for adaptive online tuning based on a multi-model algorithm. This algorithm tracks the database state and performance and the agent uses a set of pre-trained models and dynamically creates new models to tune the knobs if required. The performance measures (latency, throughput) in this case can be computed using database views such as *pg_stat_activity* in PostgreSQL.

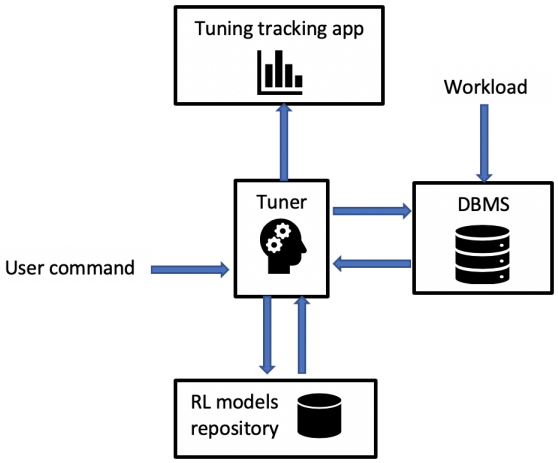


Figure 3: The multi-model database online tuning system.

The main components of our system described in Figure 3 are the tuner and the RL model repository. The tuner refers to the DDPG reinforcement learning algorithm presented in Section 3. The repository of model contains a collection of RL models, pre-trained on workloads that are as similar as possible to the workloads in the deployment environment, as well as models that are created during online tuning. The models in this repository are persisted with their weights, replay buffer, and a log file that contains the knob settings that resulted in the best database performance during training. In addition, each model is accompanied by a workload representation vector that allows to retrieve the most similar models in Algorithm 1. The last component is a web-based app we implemented using Bokeh [3] that uses the log files created during training to display in real-time the database performance measures and the values of the knobs being tuned.

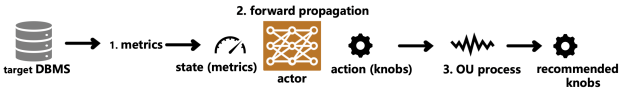


Figure 4: Knobs recommendation in deployment environment using DDPG. 1. The agent receives state and performance metrics. 2. The actor network predicts an action based on this input. 3. Exploration noise is added to the action using the Ornstein-Uhlenbeck process and a new knobs configuration is recommended.

4.1 Workload representation vectors

To generate the workload representation vectors we used an autoencoder neural network. We generated training data of state vectors that combine the database state metrics, latency, throughput, and queries per second (qps): $V = (M_0, M_1, \ldots, M_i, T, L, Q)$. These vectors were collected at different time points and for different workloads. Then, we reduced their dimension by training a simple 3-layer autoencoder and used the compressed representation from the hidden layer as the new state vectors. Since these vectors were often sparse, this procedure allowed us to obtain a

compact workload representation. The autoencoder architecture is shown in Figure 5.

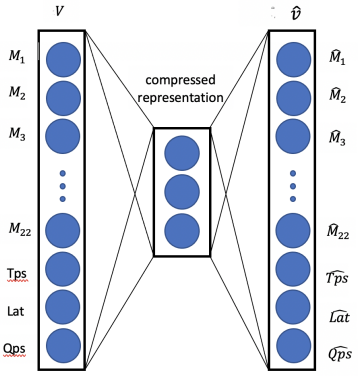


Figure 5: Workload representation dimensionality reduction using an autoencoder. V and $\hat{V}$ represents the original and reconstructed vectors, respectively. We used a 25-dimensional state vector compressed to a three-dimensional representation.

4.2 Multi-model tuning algorithm

For a given workload, Algorithm 1 selects from the N models in the repository, M_i, the ones that have high cosine similarity between the workload representation vector of the current workload, V_w and the vectors persisted with the models, V_i, and form a set of models, S. The similarity threshold is denoted by T, and was set empirically to $T = 0.8$.

Given $\tilde{N}$ models in S, we use an algorithm denoted as *Best(S)* to select the model for recommending the knob settings in every tuning iteration. The model that has the highest probability for increasing the reward (improving the database performance) is chosen to recommend the knob settings. Model selection probabilities are assigned by generating random numbers from the Beta distribution:

$$f(x; \alpha, \beta) = Cx^{\alpha-1}(1-x)^{\beta-1} \quad (2)$$

where C is a normalization constant and α, β are the distribution shape parameters. For values of $\alpha = \beta$, the distribution is symmetric and the mean is at the center of the distribution. When $\alpha > \beta$, the mean moves to the right side of the axis, and the random generator has a higher probability of yielding values larger than 0.5. We use α and β to compare the performance of the models, such that the model that performs better, has a higher α value, and therefore a higher probability to be selected in the next tuning iteration. When comparing two models, α and β counts the number of times each model produced a reward higher than the average reward from the beginning of the workload cycle to the point of measurement.

The models from the repository would always compete against a model trained from scratch (fresh model) to guarantee that the best model is being selected in cases where the model retrieved from the repository does not perform optimally. If a model M from the repository was selected and fine-tuned during the online tuning process, its fine-tuned version M' would be persisted instead of model M. The process continues as long as the online tuning phase runs. If a workload shift is detected, the model that was selected the highest number of times to predict the action within the workload cycle is persisted in the models' repository. A model selected from the repository and updated during training will be persisted with the compressed workload representation vector, the updated weights, the replay buffer experience, and the best knob settings.

Algorithm 1 Adaptive multi-model algorithm for online tuning

```
while Tune do
  S = []
  for i=0 to N do
    if cos(V_i, V_w) > T then
      S.append[M_i]
    end if
  end for
  S.append[M_new]
  while True do
    Compute a_t using Best(S) and apply it
    Wait
    Collect s_{t+1} and r_t and update the models in S
    if workload shift then
      Persist Best(S)
      Break
    end if
  end while
end while
```

5 EXPERIMENTS

In the following section, we evaluate the offline tuning algorithm and our online tuning algorithm on PostgreSQL. In the offline phase, we use the Sysbench benchmarking tool [6] to submit queries and measure the throughput and latency. Sysbench is a multi-threaded configurable benchmarking tool for OLTP workloads, where the major ones are OLTP Read/Write (R/W), OLTP Read-Only (R/O) and OLTP Write-Only (W/O). The workloads are composed of SELECT, INSERT, DELETE and UPDATE queries, where the number and mixture of the queries in each workload can be modified via command line or by modifying a Lua script that defines the workload. In addition, Sysbench allows the user to control the benchmarking duration time, the size of the database, and the number of workers.

5.1 Single-model: Static workload

In this case, a single RL model is trained and is responsible for tuning the database, regardless of the workload that is being submitted to the database. This approach works well if the database environment is relatively stable and workloads are not changing. However, if workloads are changing, the model needs to adapt to a new workload when a workload shift occurs, and it uses its past experience to recommend knobs for the new workload.

In the first experiment, we demonstrate the performance of the RL agent using the DDPG algorithm described in Figure 2. This experiment demonstrates offline training using OLTP R/W workload. We ran the algorithm for 150 episodes with the main hyperparameters setup as follows: $\gamma = 0$ (discount factor), $\sigma = 0.2$ (OUProcess noise variance), and replay buffer sampling batch size of 32 samples. To represent the DBMS state we picked 22 PostgreSQL metrics from 3 different views that provide statistics at the instance and database levels: *pg_stat_bgwriter*, *pg_stat_database*,

and *pg_stat_database_conflicts*. These views monitor various database elements such as checkpoints, buffers, deadlocks, and transactions' activity. To get the reward in the offline tuning stage, we used the throughput and latency calculated by Sysbench.

Based on experiments with different number of knobs, and expert blogs on tuning PostgreSQL (e.g., [13]), we have selected for tuning 16 knobs that had the most impact on the database performance. These knobs control various aspects of the database, such as working memory (e.g., *work_mem*, *maintenance_work_mem*), checkpoints (e.g., *checkpoint_segments*, *checkpoint_timeout*), deadlocks (*deadlock_timeout*), and auto_vaccum (*autovacuum_cost_delay*, *vacuum_cost_limit*). They do not require database restart to be updated, one of the criteria for selecting them.

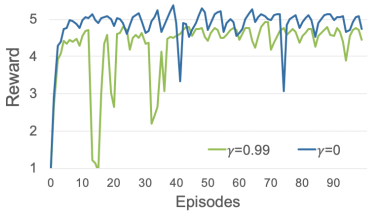
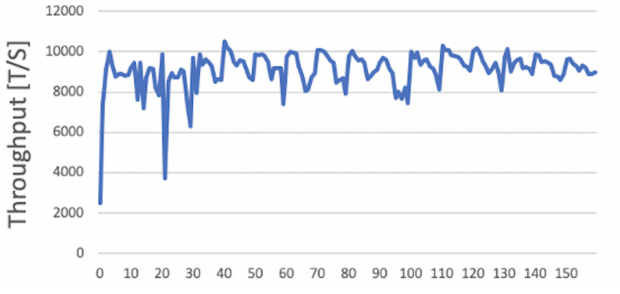
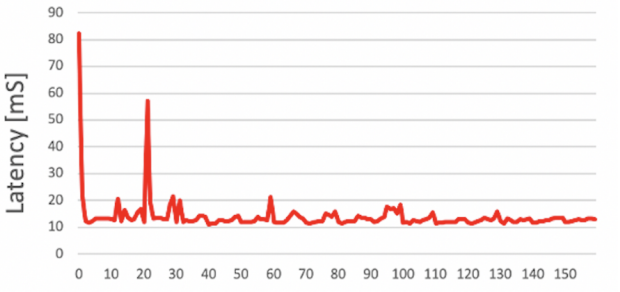


Figure 6: Reward function comparison for the OLTP R/W workload experiment. The green reward curve was obtained using the CDBTune reward [18] with $\gamma = 0.99$. The blue reward curve was obtained using our simplified reward function with $\gamma = 0$.



(a) Throughput



(b) Latency

Figure 7: Offline training: Throughput and latency plots of a PostgreSQL database tuned on an OLTP R/W workload.

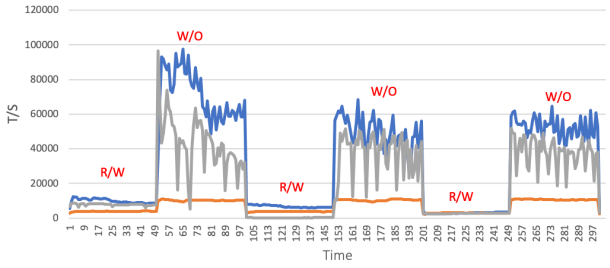
In all the experiments described in this section, the tuning starting point was the database default configuration. As Figure 7 shows, the default performance for OLTP R/W was a throughput of approximately 2000 TPS and a latency of 80 ms. As the agent explored different knob settings, we observed a 5x improvement in throughput and 8x improvement in latency. In the first iterations, the performance significantly oscillated, but the magnitude of the oscillation decreased as the agent learned the right policy for tuning the knobs. In the OLTP R/W experiment, we also compared the simplified formulation of the reward function (Eq. 1) to the formulation and γ parameter setup in [18] and observed that the simplified formulation with $\gamma = 0$ resulted in a more stable learning curve and a higher reward value (better database performance). The reward curves of the first 100 iterations are shown in Figure 6.

The offline tuning process can be repeated with any workload, such as OLTP W/O, and the models trained in the offline training phase can be used as initial pre-trained models in the multi-model online tuning experiment we describe next.

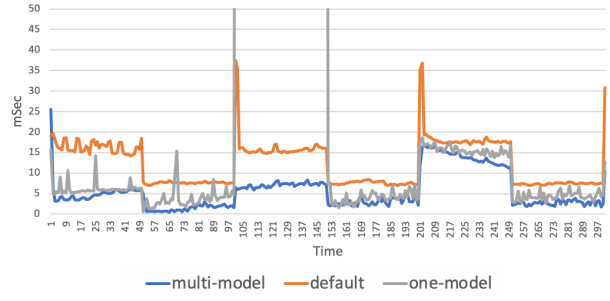
5.2 Single and multi-model approach: Changing workloads

In this experiment, we used the same knobs selected for tuning in the offline training experiment and the same 22 database metrics were used as state indicators. We created an environment in which two alternating Sysbench workloads are submitted to the database: R/W and W/O workloads and we compared the single-model, multi-model, and default database performance over 3 alternating workload cycles. At each time point in Figure 8, a workload is submitted, the state and performance metrics are collected, the RL models are updated, and finally, a knob changing action is computed by the agent to update the knobs using Algorithm 1. Each time point takes approximately 60 seconds to complete, hence, workloads are changing approximately every 60 minutes. We used a three-dimensional compressed representation of the workload vectors, obtained by the autoencoder, to detect workload changes by measuring the distance between vectors at adjacent time points and identifying large deviations. This representation was also used to select the most similar models using cosine similarity. The knob settings saved with the most similar model were applied to the database and used as a starting point for tuning.

As shown in Figure 8, the multi-model algorithm performed far better than the baseline default configuration performance, increasing the throughput of the R/W workload by a factor 2x and the throughput of the W/O by a factor of 5x. The single-model approach, on the other hand, produced inferior performance compared to the multi-model approach and was less stable. In the second cycle, for instance, presumably due to bad knob settings, the throughput dropped to approximately 300 TPS, and the latency jumped to very high values. We assume that this is related to the fact that the replay experience (state, action, reward transitions) collected in the first cycle could not be properly leveraged by the agent for learning a good configuration in the next cycle. This could be related to the fact that the single model approach uses a single replay buffer for multiple workloads, therefore, experiences from different workloads with different reward scales are mixed, and similar experiences are mapped to completely different rewards. Therefore, the agent cannot learn a good policy for tuning the knobs. In the multi-model approach, each model was trained on a different workload and used experiences unique to the workload it was trained on. This models' separation helps the agent learn the right policy for a particular workload.



(a) Throughput



(b) Latency

Figure 8: Online tuning: Throughput and latency plots of a PostgreSQL database in changing workloads environment. The tuning results were obtained using the one- and multi-model tuning algorithms, and the default knob settings. The workloads alternate between OLTP R/W and OLTP W/O.

A potential pitfall of the multi-model algorithm is the fact that many new models can be created and lead to models' explosion and a large number of competing models for every workload. However, what we observed in our experiments, is that the pre-trained models were superior to newly created models, and therefore, new models were not persisted. Therefore, we believe that eventually, the number of models in the repository would be similar or very close to the number of different workloads the agent is tuning.

6 DISCUSSION

Automatically tuning a database on-premise or in cloud environment using reinforcement learning, or a different AI technique, poses multiple challenges. One of these challenges is the fact that the database environment is constantly changing, often in the course of the day. Changes in the database size, available system resources, and workloads may affect the performance of the database at any given time point and require an adaptive algorithm that is able to sense these changes and yield optimal performance for a given environment state. In this work, we explored the effect of changing workloads on database performance and proposed an adaptive algorithm that leverages multiple DDPG reinforcement learning models to optimize the performance for each workload. Preliminary results presented in this paper on a PostgreSQL database showed that the multi-model approach has an advantage over the single-model approach in which one model is continuously trained and needs to adapt to new workloads, similar to the approach presented in [18]. Using the multi-model approach, the algorithm was able to utilize past experiences from models trained on similar workloads and improved the database default configuration throughput by a factor of 2x when tested on an OLTP R/W workload and a factor of 5x when tested on an OLTP W/O workload.

In future work, we will explore other workload types (e.g., OLAP), and we will address the problem of fluctuating performance due to other factors, such as resource elasticity.

REFERENCES

[1] Alibaba. 2020. Alibaba self-driving database. https://www.alibabacloud.com/product/das/

[2] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. 2016. *Regularized Cost-Model Oblivious Database Tuning with Reinforcement Learning.* Springer Berlin Heidelberg, Berlin, Heidelberg, 96–132. https://doi.org/10.1007/978-3-662-53455-7_5

[3] Bokeh Development Team. 2020. Bokeh: Python library for interactive visualization. https://bokeh.org/

[4] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288. https://doi.org/10.14778/2732240.2732246

[5] S. Duan, Vamsidhar Thummala, and S. Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2 (2009), 1246–1257.

[6] Alexey Kopytov. 2020. sysbench. https://github.com/akopytov/sysbench/

[7] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[8] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning.. In *ICLR*, Yoshua Bengio and Yann LeCun (Eds.).

[9] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management - aiDM'18* (2018). https://doi.org/10.1145/3211954.3211957

[10] Ryan Marcus and Olga Papaemmanouil. 2018. Towards a Hands-Free Query Optimizer through Deep Learning. arXiv:cs.DB/1809.10212

[11] Oracle. 2020. Oracle autonomous database. https://www.oracle.com/autonomous-database/

[12] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. arXiv:cs.DB/1803.08604

[13] Percona. 2018. Tuning PostgreSQL. https://www.percona.com/blog/2018/08/31/tuning-postgresql-database-parameters-to-optimize-performance/

[14] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. arXiv:cs.DB/1801.05643

[15] G. E. Uhlenbeck and L. S. Ornstein. 1930. On the Theory of the Brownian Motion. *Phys. Rev.* 36 (Sep 1930), 823–841. Issue 5. https://doi.org/10.1103/PhysRev.36.823

[16] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17).* 1009–1024. https://db.cs.cmu.edu/papers/2017/p1009-van-aken.pdf

[17] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Farooq Minhas, Umar, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (May 2020).

[18] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19).* Association for Computing Machinery, New York, NY, USA, 415–432. https://doi.org/10.1145/3299869.3300085

Short Paper

Optimising Fairness Through Parametrised Data Sampling

Vladimiro González-Zelaya
Newcastle University, UK
Universidad Panamericana, Mexico
Escuela de Ciencias Económicas y Empresariales
cvgonzalez@up.edu.mx

Julián Salas
Universitat Rovira i Virgili, Spain
Department of Computer Engineering and Mathematics
Center for Cybersecurity Research of Catalonia, Spain
julian.salas@urv.cat

Dennis Prangle
Newcastle University, UK
School of Mathematics, Statistics and Physics
dennis.prangle@ncl.ac.uk

Paolo Missier
Newcastle University, UK
School of Computing
paolo.missier@ncl.ac.uk

ABSTRACT

Improving machine learning models' fairness is an active research topic, with most approaches focusing on specific definitions of fairness. In contrast, we propose ParDS, a parametrised data sampling method by which we can optimise the *fairness ratios* observed on a test set, in a way that is agnostic to both the specific fairness definitions, and the chosen classification model. Given a training set with one binary protected attribute and a binary label, our approach involves correcting the positive rate for both the *favoured* and *unfavoured* groups through resampling of the training set. We present experimental evidence showing that the amount of resampling can be optimised to achieve target fairness ratios for a specific training set and fairness definition, while preserving most of the model's accuracy. We discuss conditions for the method to be viable, and then extend the method to include multiple protected attributes. In our experiments we use three different sampling strategies, and we report results for three commonly used definitions of fairness, and three public benchmark datasets: *Adult Income*, *COMPAS* and *German Credit*.

1 INTRODUCTION

The increasing presence of automated decisions in our lives has led to a rising concern about the way in which these decisions are taken, spurring research into the *fairness* of predictive models. These models are often learnt from biased data, reflecting historical disparities and discrimination [27]. We propose ParDS, a fairness-definition and classifier agnostic resampling method, which may be easily implemented on top of existing ML solutions and can satisfy specific classification model requirements. ParDS is modulated through the continuous parameter d, which determines the amount of resampling introduced into the training data, and has two possible use cases: to find the optimal amount of correction for a specific fairness/classifier combination and to control a classifier's fairness/accuracy trade-off.

Standard data preparation techniques may be used to correct the fairness behaviour of a classification model [29]. ParDS is based on data resampling, which is well understood and part of the typical data management pipeline [23]. Being a preprocessing operator, ParDS may easily be incorporated along data cleaning into existing database solutions. Like other resampling techniques, ParDS can be computationally inexpensive and yield reduced classifier learning times, as shown in Subsection 4.2.

Our method offers the versatility of using both generic (random undersampling, random oversampling and *SMOTE* [6]) and fairness-specific (preferential sampling [20]) methods.

Multiple definitions of fairness have been proposed [25], which are sometimes in contrast with one another. A decision rule that satisfies one of the definitions may well prove to be very unfair for a different one [10]. For example, determining university admissions through gender quotas may achieve *demographic parity*, but it makes the acceptance rates for good students of different genders disparate.

A common resampling problem is the loss of predictive accuracy caused by such interventions [3]. In our setting, such loss can also be controlled through parameter d, allowing for a decision in the amount of accuracy/fairness trade-off the user is willing to accept. Furthermore, our experiments in Section 4 show that even at high correction levels, the accuracy loss for ParDS is relatively low.

1.1 Related Work

A classifier's fairness may be corrected by *preprocessing* the training data, *in-processing* the learning algorithm [1, 5, 30–32] or *post-processing* a classifier's predictions [18]. Our method belongs to the group of preprocessing solutions.

Fairness-aware preprocessing is defined [14] as a set of techniques that modify input data so that any classifier trained on such data will be fair. There are four main ways in which to make appropriate adjustments to data in order to enforce fairness [21]: suppressing certain features, also known as *fairness through unawareness (FTU)* [15], *massaging* variable values [4, 9, 13], reweighing features [19, 24], and resampling data instances [8, 20, 28, 29].

Data resampling, the category of ParDS, is less invasive in nature than *FTU* or *massaging*, since the original data is preserved and only the frequency with which the instances are represented is modified. In contrast, *FTU* disposes of large amounts of data without a guarantee on the effect of said intervention and *massaging* effectively creates synthetic data, which does not necessarily reflect the ground truth.

Preferential Sampling (PS) [20] is a similar method to ParDS, in the sense that it resamples the favoured/unfavoured and positive/negative combinations separately in order to equalise the favoured and unfavoured groups' positive ratios. We empirically show that the optimal fairness correction depends on the selected sampling method, classifier and fairness definition. Equalising the positive ratios across protected attribute groups is not necessarily the best approach, hence we modulate our corrections via

parameter d. When using *PS* to resample, ParDS generalises it, with the unmodified *PS* corresponding to the $d = 0$ case.

SMOTEBoost [8] oversamples the minority group through synthetic data based on real data instances, with a focus on improved minority predictions and indirectly improving fairness.

Other related methods include *Capuchin* [29], a causal-fairness centric, non-parametrised resampling method and Feldman et al. [13], a *massaging* method where parameter λ is used to create linear interpolations of the original dataset and a *repaired* copy to find the optimal combination.

1.2 Contributions

We introduce ParDS, a parametrised resampling-based fairness-correcting method. ParDS is fairness-definition and classifier agnostic, and achieves close to optimal fairness correction with a small loss in predictive performance. We present extensive experiments to benchmark the effectiveness of the method using the *Adult Income*, *COMPAS* and *German Credit* datasets, and our implementation is available as a collection of *Jupyter Notebooks* at https://github.com/vladoxNCL/fairCorrect. This is a substantial extension of our preliminary workshop paper [17], presented at the *2019 KDD XAI Workshop*. Its additional contributions are four-fold:

(1) We estimate the optimal fairness correction using Bayesian optimisation.
(2) We present experimental evidence on synthetic datasets of our method's viability and effectiveness with respect to the linear separability of the training set.
(3) We make an initial investigation into extending the method to multiple protected attributes.
(4) We benchmark and compare our work with several existing fairness-correction methods.

2 DEFINITIONS

We will say a binary classifier's label can be *positive* or *negative* referring to the desirable and non-desirable outcome of a prediction, respectively.

A dataset's *protected attribute (PA)* refers to a variable that may be object of discrimination, due to historical bias or otherwise. In our particular case we will be dealing with a single binary PA.

We will call the ratio of the number of positive instances divided by the total number of instances in a specific group the *positive ratio (PR)* of the group.

Among the two PA groups, the one having the highest PR will be referred to as the *favoured* group F, while the other one will be referred to as the *unfavoured* group U. When required, we will refer to the positive and negative instances of F and U as F^+, F^-, U^+ and U^-, respectively.

We based our analyses on three ratios, *Demographic Parity Ratio (DPR)*, *Equality of Opportunity Ratio (EOR)* and *Proxy Fairness Ratio (PFR)*, associated to their respective fairness definitions [22, 25]. In these definitions, the positive label is identified with $Y = 1$ and the negative label with $Y = 0$.

Definition 2.1 (Fairness Ratios).

$$DPR := \frac{\mathbb{P}(\hat{Y} = 1 \mid \text{PA} = U)}{\mathbb{P}(\hat{Y} = 1 \mid \text{PA} = F)},$$

$$EOR := \frac{\mathbb{P}(\hat{Y} = 1 \mid \text{PA} = U, Y = 1)}{\mathbb{P}(\hat{Y} = 1 \mid \text{PA} = F, Y = 1)},$$

$$PFR := \frac{\mathbb{P}(\hat{Y} = 1 \mid do(\text{PA} = U))}{\mathbb{P}(\hat{Y} = 1 \mid do(\text{PA} = F))}.$$

For *DPR* and *EOR*, we evaluate the ratio of the positive classification probabilities for U and F. *PFR* is computed by intervening on the test set T twice, assigning every individual in T the PA-values U and F, resulting in $T_{\text{PA}=U}$ and $T_{\text{PA}=F}$, respectively. We then evaluate the quotient of the intervened sets' classification *PRs*; in all cases, the ratios quantify how close the classifier comes to optimal fairness.

3 METHODOLOGICAL APPROACH

We have focused on datasets with both binary protected attributes and labels. The plots in this section result from applying ParDS to the *Adult Income (Income)* dataset [12].

We introduce the *disparity correction* parameter $d \in [-1, 1]$, which may be used for two different objectives:

- To modulate a classifier's fairness/accuracy trade-off.
- To optimise a classifier with respect to a fairness definition.

Our main objective will be the third one, to estimate the d-value optimising a classifier's predictions with respect to a fairness definition. We summarise the method as follows:

(1) Define PR-correcting functions for F and U.
(2) Select a *sampling strategy* to correct the training set.
(3) Estimate the fairness-specific optimal d-value.

Details on each of these steps now follow.

3.1 Parametrising Correction

The first step is to define linear functions that will yield corrected PRs for both PA groups. These functions, which we will call $f^+(d)$ and $u^+(d)$, should satisfy the constraints: $f^+(1) = \text{PR}(F)$, $f^+(-1) = \text{PR}(U)$ and $u^+(d) = f^+(-d)$.

The equations for these two linear functions are

$$f^+(d) = md + b, \quad u^+(d) = -md + b,$$

with coefficients

$$m = \frac{\text{PR}(F) - \text{PR}(U)}{2}, \quad b = \frac{\text{PR}(F) + \text{PR}(U)}{2}.$$

3.2 Sampling Strategies

In the second step, we use the resulting corrected ratios $f^+(d)$ and $u^+(d)$ to produce a resampled training set $\{\hat{U}, \hat{F}\}$ satisfying these ratios. The required amount of resampling for F and U will depend on d and the selected strategy.

ParDS can use one of four different sampling methods, modified to work on specific PA-label subgroups: random undersampling (*Under*), random oversampling (*Over*), *SMOTE* [7] and preferential sampling (*PS*) [20]. Depending on the sampling method, the following subgroups will be modified:

***Under*:** Undersample F^+ and U^-.
***Over*:** Oversample F^- and U^+.
***SMOTE*:** Oversample F^- and U^+.
***PS*:** Undersample F^+ and U^-, oversample F^- and U^+.

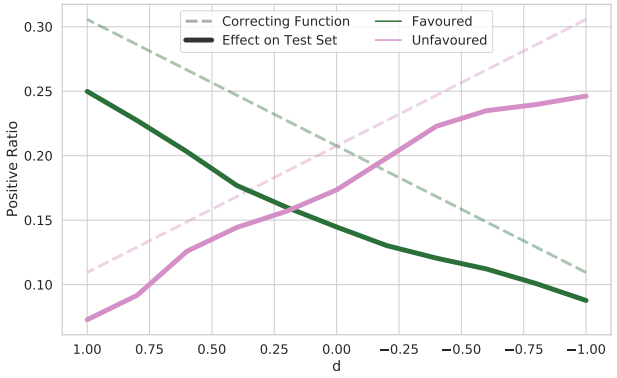


Figure 1: Correcting functions $f^+(d)$ and $u^+(d)$ applied to *Income* and their effect on the test set. The d-axis is reversed, going from 1 (no correction) to -1 (maximum correction). Note that the test-set PRs do not intersect at $d = 0$.

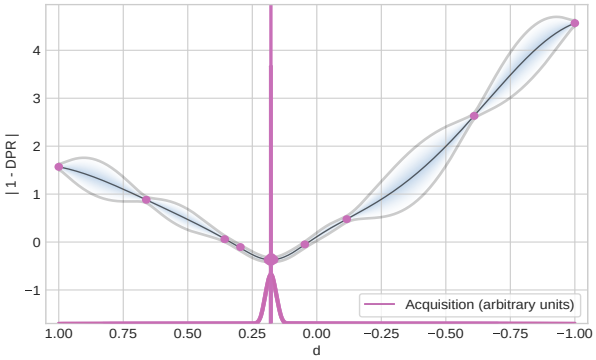


Figure 2: Plot of *GPyOpt*'s approximation of *DPR* as function of d for *Income*. The bottom red curve displays the resulting distribution for the optimal d-value.

The resampled $\hat{F}$ must satisfy

$$\frac{|\hat{F}^+|}{|\hat{F}^+| + |\hat{F}^-|} = f^+(d),$$

which may be rewritten as

$$\frac{|\hat{F}^+|}{|\hat{F}^-|} = \frac{f^+(d)}{1 - f^+(d)}. \tag{1}$$

The selected strategy will determine whether F^+ or F^- will be resampled to satisfy (1). Using *Under*, for example, $\hat{F}^+$ results from undersampling F^+, while $\hat{F}^- = F^-$. In contrast, using *Over* produces $\hat{F}^-$ from oversampling F^- while $\hat{F}^+ = F^+$. An analogous equation to (1) is used to resample U onto $\hat{U}$.

After the training-set has been resampled, a classifier learnt from the corrected training-set will display an improvement in fairness with respect to a classifier learnt from the original data. An example of the produced PR-correcting functions and their effect over *Income* is shown in Figure 1.

3.3 Finding the Optimal Amount of Sampling

Finally, the third step is to estimate the optimal correction for a specific fairness definition. As classification algorithms usually display non-linear—and sometimes unexpected—behaviours, it is not possible to deduce a closed-form solution to this optimisation problem. Hence, it becomes necessary to numerically approximate a solution.

A naïve approach is to compare the resulting fairness ratios for different values of d, and select the one producing the ratio closest to 1. As we will see on Section 4.2.1, it is easy to find d-values close to the optimal by trial and error, yet this optimal d-value will usually be different for distinct fairness definitions.

A more systematic way to approximate the optimal value of d is to use Bayesian optimisation [16]. This technique estimates the objective function on candidate values obtained from previous function estimations. The main reasons for choosing Bayesian over other optimisation methods are that every fairness ratio evaluation will be different due to the randomness in the sampling process and that Bayesian optimisation is good when estimating the objective function is expensive, e.g. our setting, since we work on large datasets and take the average over many estimations.

We have implemented a simple fairness optimiser using the *GPyOpt* [2] package, with a standard Gaussian process using d as the only parameter and the distances of the different fairness ratios to 1 as objective functions, e.g. estimate the d-value yielding the fairest *DPR* expectation:

$$\arg\min_d |1 - \mathbb{E}[DPR(d)]| \quad \text{subject to} \quad -1 \leq d \leq 1.$$

An example run, used to approximate the optimal correction for Demographic Parity on *Income* may be seen in Figure 2.

3.4 Multiple Protected Attributes

We have generalised ParDS to multi-class PAs, as well as to multiple PA variables. In some cases, this could be addressed by binning several PA labels into just two categories, u and f. However, these arbitrary assignments would imply a loss of granularity in any subsequent fairness analysis. As an alternative, we have chosen to consider a *combined* PA, which may be obtained for every datapoint $p \in$ train as follows:

(1) Evaluate $\mathrm{PR}(D)$ for the training set D.
(2) Define a set of PAs: $\{\mathrm{PA}_1, \mathrm{PA}_2, \ldots, \mathrm{PA}_k\}$.
(3) Evaluate

$$\mathrm{PR}_i(p) = \mathrm{PR}\left(\mathrm{PA}_i(p)\right) - \mathrm{PR}(D)$$

for $i \in \{1, 2, \ldots, k\}$.
(4) Aggregate the partial PRs to obtain a combined value

$$\mathrm{PR}^*(p) = \sum_{i=1}^{k} \mathrm{PR}_i(p).$$

(5) Define the combined PA of p as

$$\mathrm{PA}^*(p) = \begin{cases} F & \text{if } \mathrm{PR}^*(p) > 0, \\ U & \text{if } \mathrm{PR}^*(p) \leq 0. \end{cases}$$

This solution allows for a much more granular approach on determining a datapoint's relative "prosperity" with respect to every PA, as some PA attributes may prove to be more determining of disparate treatment than others, and the effects of several PAs may cancel each other out.

Our experiments, carried out on *Income*, provide positive results, as described next. Figure 3 compares the effects—for unfavoured groups of different PAs (Gender, Nationality, Race and Age)—of applying disparity correction based on a single PA (Gender) to doing it based on a combined PA aggregating Gender, Age, Race and Country for *Income*. As can be seen, when correcting

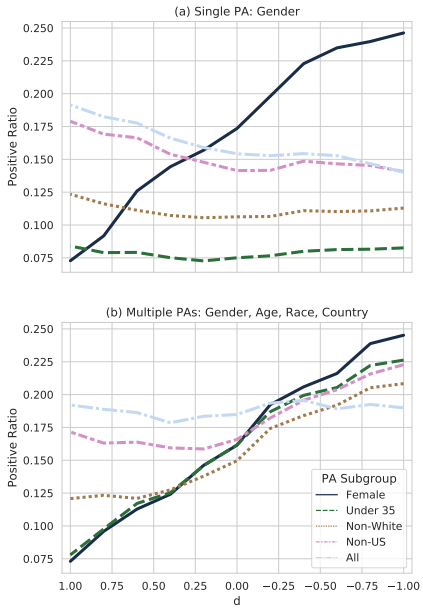


Figure 3: Positive ratios for the different PAs' unfavoured groups on *Income*, correcting for (a) one PA and (b) multiple PAs.

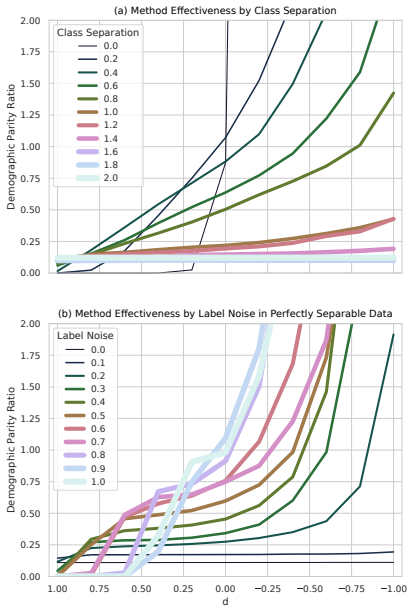


Figure 4: Correction effectiveness by separability. (a): On close-to-linearly separable data, the method becomes highly inefficient, or even stops working at all. (b): Introducing random noise into a separable dataset lets correction become effective again.

for Gender alone, the other unfavoured groups' PR remains relatively constant or gets worse. Likewise, the overall PR shows a drop on its PR as more correction is applied. When correcting for the combined PA, on the other hand, all of the unfavoured PRs improve at a similar rate, whilst the overall PR remains relatively constant across different correction levels. In short, this extension provides ParDS with the capability to correct for multiple biases simultaneously, at the individual level with similar optimal d-values across PAs. This method for combining several PAs into a single combined one, though, is not unique, and could further be improved by adding weights to the different PAs set as hyperparameters by experts.

4 EXPERIMENTAL EVALUATION

This section reports the effectiveness of ParDS regarding separability in Subsection 4.1, comparing sampling strategies and fairness definitions in Subsection 4.2 and benchmarking ParDS with existing fairness-correcting methods, in Subsection 4.3.

4.1 Separability

To verify the effect of separability on ParDS' effectiveness, we created 11 simple datasets, consisting of one continuous feature f and one binary label l. These datasets were created using the *scikit-learn*'s [11] *make_classification* function, with varying levels of class separability s, ranging from 0 (completely mixed up) to 2 (over 95% probability of complete separation). What this function does is sample feature values from normal distributions centered at s and $-s$ for the two classes, respectively. A PA was then randomly added, ensuring a fixed 50/50 proportion of F vs U datapoints, with PRs of 0.9 and 0.1 for F and U, respectively.

As may be seen in Figure 4a, the greater separability data has, the less effective our correcting method becomes (represented by a near-flat demographic parity ratio curve as a function of d). However, adding random noise to a linearly separable dataset (effectively rendering it inseparable again) restores the effectiveness of ParDS. To test this, we created a linearly separable dataset with $s = 2$, and gradually introduced noise through parameter n taking values from 0 to 1, the proportion of randomly-assigned labels. As shown in Figure 4b, this intervention can render fairness correction effective again, even with a small amount of added noise.

4.2 Method Validation

We tested ParDS on three datasets commonly used in ML fairness research literature: *Adult Income (Income)* [12], *COMPAS* [26] and *German Credit* [12].

For every dataset, we performed the following experiment 50 times, and then averaged the results for robustness:

(1) Random train/test split the data with 90/10 proportion.
(2) For Proxy Fairness checking, make two copies of the *test* set T and intervene PA as either U or F, obtaining $T_{\text{PA}=U}$ and $T_{\text{PA}=F}$, respectively.
(3) For each sampling function, obtain 11 training sets, corresponding to $d \in \{1, 0.8, 0.6, \ldots, -1\}$.
(4) For each of these training sets, fit a classifier.
(5) For every model, get predictions for T, $T_{\text{PA}=U}$ and $T_{\text{PA}=F}$.
(6) Compute metrics for accuracy, *DPR*, *EOR* and *PFR*, as well as the model coefficients.

We then proceeded to analyse the resulting fairness metrics, and compared our results with *PS*.

4.2.1 Results. As expected, fairness correction has an impact over a classifier's predictive performance. Figure 5 shows the fairness-accuracy trade-off for the different sampling strategies for the three fairness ratios over *Income*. As may be seen, the trade-off is similar across the different sampling strategies, and the loss in predictive performance for optimal fairness will be definition dependant.

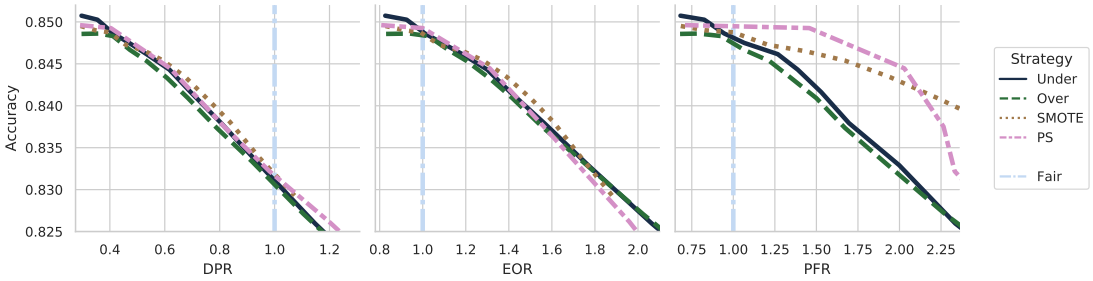


Figure 5: Fairness-accuracy trade-off for *DPR*, *EOR* and *PFR* on *Income*.

Table 1 shows diverse performance metrics for ParDS using the different sampling strategies to correct γ_{sr} on *Income*. The presented means and confidence intervals (CIs) result from 100 independent train/test splits, then using each sampling strategy with the optimal d-value for each estimated through Bayesian optimisation. As may be seen, there is a big difference in computing time across strategies, with *SMOTE* being over 10 times slower than *Under*. On the other hand, *SMOTE* produced the best scores for most performance metrics. Optimal fairness correction was achieved within the CIs for all methods, with roughly the same accuracy loss trade-off. Interestingly, running *Under* before training the classifier was 35% faster than just training the classifier over the full dataset. This would provide an additional advantage for *Under*-corrected training sets when learning models from large-scale datasets.

4.3 Comparison with Other Methods

An intrinsic advantage of ParDS is that it can optimise a classifier with respect to different group fairness definitions. Three definitions: γ_{sr} [5], *discrimination (disc)* [21] and *equalised odds (eOdds)* [19] were used for our comparisons.

Tables 2 and 3 compare ParDS with a variety of *preprocessing* [4, 8, 19, 20, 24, 29], *in-processing* [1, 5, 30–32] and *post-processing* [18] fairness-correcting methods.

Since four different classification algorithms were used on the papers we compared with—AdaBoost (AB), decision trees (DT), Gaussian naïve Bayes (GNB) and logistic regression (LR)—we present ParDS' results using all three of them. We optimised our classifiers to compare with the state-of-the-art methods, hence three of our presented methods are optimised for *DPR* and two are optimised for *eOdds*. We evaluated our metrics using the same classification algorithms as the ones used in the papers we compare with. The objective functions to optimise were $|1 - DPR|$ and $|1 - eOdds|$ for *DPR* and *eOdds*, respectively, with 0 being the best value the objective function may take in both cases.

For every tested d-value we averaged the resulting *DPR* of 50 random 90/10 train/test splits, finding optimal d-values of {0.8338, -0.1803, -1.1528, -0.6083} for AB, DT, GNB and LR, respectively. All of the classifiers were trained using the default scikit-learn hyper-parameter values; using these parameter values, we ran ParDS 10 times and averaged the resulting metrics. The fairness and accuracy metrics for the compared methods refer to the *best* reported values in [5, 19, 20, 29, 32]. Likewise, for methods evaluated on more than one classifier, we present the best one.

Our *eOdds*-optimised AdaBoost classifier produced the best overall accuracy (86%), while showing an *eOdds* value within 3% of the best performing method [24]. Regarding our *DPR* optimised classifiers, although LR performed the best overall, both ParDS' DT and GNB performed better than the other methods' DTs and GNBs, respectively. Interestingly, ParDS' LR produced the fairest classifiers with respect to definitions γ_{sr} and *disc*, even though they were actually optimised for *DPR*. While the accuracy of *DPR*-optimised ParDS LR was not the best (83%), it came within 1% of the best performing classifiers (ParDS' DT, Kamiran and Calders [20] and Zafar et al. [30], with an accuracy of 84%).

5 CONCLUSION

In this paper we define ParDS, a parametrised fairness optimisation method agnostic to both fairness definitions and classification models. Correcting through training set resampling, we have shown that ParDS produces fairness-optimal predictions with a small loss in predictive power. When compared with the existing methods, in most cases ParDS produces the best fairness performance.

In future work we intend to further improve our data resampling methods, in order to optimise for different fairness definitions at once. Although ParDS shows a relatively low impact on prediction performance and its main objective is to estimate the optimal amount of correction with respect to fairness, we would like to find a way to consider predictive performance as well, either in the form of a restriction—e.g. a maximum loss in accuracy or a minimum level of fairness—or by setting an acceptable trade-off rate between both metrics.

ACKNOWLEDGMENTS

This research was partly supported by the Spanish Government under project RTI2018-095094-B-C21 "CONSENT".

REFERENCES

[1] Alekh Agarwal, Alina Beygelzimer, Miroslav Dudík, John Langford, and Hanna Wallach. 2018. A reductions approach to fair classification. *arXiv preprint arXiv:1803.02453* (2018).
[2] The GPyOpt authors. 2016. GPyOpt: A Bayesian Optimization framework in Python. http://github.com/SheffieldML/GPyOpt.
[3] Richard Berk, Hoda Heidari, Shahin Jabbari, Michael Kearns, and Aaron Roth. 2018. Fairness in criminal justice risk assessments: The state of the art. *Sociological Methods & Research* (2018), 0049124118782533.
[4] Flavio Calmon, Dennis Wei, Bhanukiran Vinzamuri, Karthikeyan Natesan Ramamurthy, and Kush R Varshney. 2017. Optimized pre-processing for discrimination prevention. *Advances in Neural Information Processing Systems* 30 (2017), 3992–4001.
[5] L Elisa Celis, Lingxiao Huang, Vijay Keswani, and Nisheeth K Vishnoi. 2019. Classification with fairness constraints: A meta-algorithm with provable guarantees. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*. 319–328.
[6] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research* 16 (2002), 321–357.
[7] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

Table 1: Fairness and performance metrics comparison using the four sampling strategies on *Income*, optimising LR models for γ_{sr} with 95% CIs. The best score for each metric is highlighted.

	No Correction	*Under*	*Over*	*SMOTE*	*PS*
Estimated Optimal d		−0.5770	−0.6083	−0.8246	0.1784
Time (s/iteration)	1.324 ± 0.018	**0.857 ± 0.014**	1.952 ± 0.030	10.974 ± 0.085	1.237 ± 0.016
Discrimination	0.178 ± 0.003	−0.001 ± 0.002	**0.0 ± 0.003**	0.006 ± 0.003	−0.003 ± 0.003
γ_{sr}	0.295 ± 0.007	1.008 ± 0.018	**1.005 ± 0.019**	0.966 ± 0.018	1.023 ± 0.018
Accuracy	0.85 ± 0.001	0.832 ± 0.001	0.831 ± 0.001	**0.833 ± 0.001**	0.831 ± 0.001
Balanced Accuracy	0.761 ± 0.002	0.695 ± 0.002	0.692 ± 0.002	**0.717 ± 0.002**	0.712 ± 0.002
Precision	0.737 ± 0.003	**0.77 ± 0.004**	**0.77 ± 0.004**	0.727 ± 0.004	0.726 ± 0.004
Recall	0.59 ± 0.003	0.43 ± 0.004	0.425 ± 0.003	**0.493 ± 0.003**	0.482 ± 0.003
F-Score	0.655 ± 0.003	0.552 ± 0.003	0.547 ± 0.003	**0.587 ± 0.003**	0.58 ± 0.003
ROC AUC	0.761 ± 0.002	0.695 ± 0.002	0.692 ± 0.002	**0.717 ± 0.002**	0.712 ± 0.002

Table 2: Fairness metrics and accuracy comparison of our *DPR*-optimised method with related fairness-correcting methods. The best result for each metric is highlighted.

Algorithm	CLF	*disc*	γ_{sr}	Accuracy
No Correction	LR	0.18	0.295	0.85
ParDS (*Over*)	LR	**0.00**	**1.00**	0.83
ParDS (*PS*)	GNB	**0.00**	0.98	0.82
ParDS (*PS*)	DT	0.01	0.97	**0.84**
Kamiran and Calders [20]	DT	0.03	—	**0.84**
Zemel et al. [32]	LR	0.20	—	0.68
Calmon et al. [4]	LR	0.03	—	0.79
Salimi et al. [29]	MLP	0.06	—	0.79
Zafar et al. [31]	GNB	—	0.87	0.77
Hardt et al. [18]	GNB	—	0.85	0.81
Zafar et al. [30]	GNB	—	0.42	**0.84**
Agarwal et al. [1]	GNB	—	0.72	0.79
Celis et al. [5]	GNB	—	0.95	0.77

Table 3: Equalised odds, accuracy and balanced accuracy comparison of our *eOdds*-optimised method with related fairness-correcting methods. The best result for each metric is highlighted.

Algorithm	CLF	*eOdds*	Accuracy
No Correction	AB	0.18	0.86
ParDS (*PS*)	AB	0.08	**0.86**
ParDS (*PS*)	LR	0.09	0.83
Krasanakis et al. [24]	LR	**0.05**	0.82
Iosifidis and Ntoutsi [19]	AB	0.08	0.83
Chawla et al. [8]	AB	0.47	0.81

[8] Nitesh V Chawla, Aleksandar Lazarevic, Lawrence O Hall, and Kevin W Bowyer. 2003. SMOTEBoost: Improving prediction of the minority class in boosting. In *European conference on principles of data mining and knowledge discovery*. Springer, 107–119.

[9] Silvia Chiappa and Thomas PS Gillam. 2018. Path-specific counterfactual fairness. *arXiv preprint arXiv:1802.08139* (2018).

[10] Alexandra Chouldechova. 2017. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big data* 5, 2 (2017), 153–163.

[11] Sci-kit Learn Developers. 2019. scikit-learn: machine learning in Python.

[12] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[13] Michael Feldman, Sorelle A Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and removing disparate impact. In *proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 259–268.

[14] Sorelle A Friedler, Carlos Scheidegger, Suresh Venkatasubramanian, Sonam Choudhary, Evan P Hamilton, and Derek Roth. 2019. A comparative study of fairness-enhancing interventions in machine learning. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*. ACM, 329–338.

[15] Pratik Gajane and Mykola Pechenizkiy. 2017. On formalizing fairness in prediction with machine learning. *arXiv preprint arXiv:1710.03184* (2017).

[16] Javier González, Michael Osborne, and Neil D Lawrence. 2016. GLASSES: Relieving the myopia of Bayesian optimisation. (2016).

[17] Vladimiro González-Zelaya, Paolo Missier, and Dennis Prangle. 2019. Parametrised Data Sampling for Fairness Optimisation. (2019). Presented on the *2019 XAI Workshop at SIGKDD, Anchorage, AK, USA*. Available at http://homepages.cs.ncl.ac.uk/paolo.missier/doc/kddSubmission.pdf.

[18] Moritz Hardt, Eric Price, Nati Srebro, et al. 2016. Equality of opportunity in supervised learning. In *Advances in neural information processing systems*. 3315–3323.

[19] Vasileios Iosifidis and Eirini Ntoutsi. 2019. AdaFair: Cumulative fairness adaptive boosting. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 781–790.

[20] Faisal Kamiran and Toon Calders. 2010. Classification with no discrimination by preferential sampling. In *Proc. 19th Machine Learning Conf. Belgium and The Netherlands*. Citeseer, 1–6.

[21] Faisal Kamiran and Toon Calders. 2012. Data preprocessing techniques for classification without discrimination. *Knowledge and Information Systems* 33, 1 (2012), 1–33.

[22] Niki Kilbertus, Mateo Rojas Carulla, Giambattista Parascandolo, Moritz Hardt, Dominik Janzing, and Bernhard Schölkopf. 2017. Avoiding discrimination through causal reasoning. In *Advances in Neural Information Processing Systems*. 656–666.

[23] SB Kotsiantis, Dimitris Kanellopoulos, and PE Pintelas. 2006. Data preprocessing for supervised leaning. *International Journal of Computer Science* 1, 2 (2006), 111–117.

[24] Emmanouil Krasanakis, Eleftherios Spyromitros-Xioufis, Symeon Papadopoulos, and Yiannis Kompatsiaris. 2018. Adaptive sensitive reweighting to mitigate bias in fairness-aware classification. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 853–862.

[25] Matt J Kusner, Joshua Loftus, Chris Russell, and Ricardo Silva. 2017. Counterfactual fairness. In *Advances in Neural Information Processing Systems*. 4066–4076.

[26] Jeff Larson, Surya Mattu, Lauren Kirchner, and Julia Angwin. 2016. How we analyzed the COMPAS recidivism algorithm. *ProPublica (5 2016)* 9 (2016).

[27] David Madras, Elliot Creager, Toniann Pitassi, and Richard Zemel. 2019. Fairness through causal awareness: Learning causal latent-variable models for biased data. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*. 349–358.

[28] Donald B Rubin. 1973. The use of matched sampling and regression adjustment to remove bias in observational studies. *Biometrics* (1973), 185–203.

[29] Babak Salimi, Luke Rodriguez, Bill Howe, and Dan Suciu. 2019. Interventional fairness: Causal database repair for algorithmic fairness. In *Proceedings of the 2019 International Conference on Management of Data*. 793–810.

[30] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P Gummadi. 2017. Fairness beyond disparate treatment & disparate impact: Learning classification without disparate mistreatment. In *Proceedings of the 26th international conference on world wide web*. 1171–1180.

[31] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P Gummadi. 2015. Fairness constraints: Mechanisms for fair classification. *arXiv preprint arXiv:1507.05259* (2015).

[32] Rich Zemel, Yu Wu, Kevin Swersky, Toni Pitassi, and Cynthia Dwork. 2013. Learning fair representations. In *International Conference on Machine Learning*. 325–333.

Short Paper

Using Landmarks for Explaining Entity Matching Models

Andrea Baraldi, Francesco Del Buono, Matteo Paganelli, Francesco Guerra
UNIMORE-DIEF
Modena, Italy
firstname.lastname@unimore.it,andrea.baraldi96@unimore.it

ABSTRACT

The state of the art approaches for performing Entity Matching (EM) rely on machine & deep learning models for inferring pairs of matching / non-matching entities. Although the experimental evaluations demonstrate that these approaches are effective, their adoption in real scenarios is limited by the fact that they are difficult to interpret. Explainable AI systems have been recently proposed for complementing deep learning approaches. Their application to the scenario offered by EM is still new and requires to address the specificity of this task, characterized by particular dataset schemas, describing a pair of entities, and imbalanced classes.

This paper introduces Landmark Explanation, a generic and extensible framework that extends the capabilities of a post-hoc perturbation-based explainer over the EM scenario. Landmark Explanation generates perturbations that take advantage of the particular schemas of the EM datasets, thus generating explanations more accurate and more *interesting* for the users than the ones generated by competing approaches.

1 INTRODUCTION

Despite the effort put in the past 30 years, Entity Matching (EM), the task that identifies data items that refer to the same real-world entity, is still an open challenge. State of the art approaches (e.g., DeepER [7], DeepMatcher [12], DITTO [10], and many others [2, 19]), based on Machine Learning (ML) and Deep Learning (DL) models, have been demonstrated to be effective in the experimental datasets. Nevertheless, their adoption in real business scenarios is hampered by several factors, including the need for large amounts of training data, the need for expert users for the configuration of their hyper-parameters and the inability to easily interpret how the models make their decisions.

Explaining the behavior of ML and DL models is now a challenging research topic [5]. Its application to EM could facilitate the adoption of EM techniques in business scenarios. An improved ability to interpret the models would increase 1) user confidence in the adoption of ML and DL techniques, 2) the ability to debug erroneous behaviors and diagnose unexpected results, and 3) improve the functionality of the approaches. Moreover, it would decrease the need for domain experts to evaluate the effectiveness of EM approaches, task that is typically executed through manual, expensive, and time-consuming processes.

Although several explanation systems have already been proposed in the literature (e.g., LIME [14], Shapley [8], Anchor [15], and Skater[1]) , their application to EM tasks is not straightforward and only few approaches have partially addressed it [4, 6, 11, 17].

[1]https://github.com/oracle/Skater

left_name	right_name	left_description	right_description	left_price	right_price
sony digital camera with lens kit dslra200w	nikon digital camera leather case 5811	sony alpha digital slr camera with lens kit dslra200w 10.2 megapixels	leather black	849.99	7.99

Figure 1: Example of EM record to explain.

The main motivation is that EM is conceived by ML and DL systems as a binary classification problem, where the class shows if the pairs of entities described in the dataset records are matching, and the dataset entry is composed of pairs of attributes describing the same feature of different entities. This structure is "unusual" in ML and DL, where the records conversely describe single evidence. Moreover, the datasets are usually imbalanced: the number of records belonging to the matching class is far less than the non matching ones. Finally, the attributes describing the same features of different entities have close statistical distributions (or close word distributions in case of categorical attributes) even when they refer to different entities.

In this paper, we present Landmark Explanation a system for explaining EM model predictions, that extends the capabilities of a post-hoc perturbation-based local explainer over this specific scenario. Post-hoc perturbation-based explainers analyze the records to explain and build a surrogate linear model where the features are the tokens (e.g., the words in case of textual attributes) composing the attribute values. The explanation is directly generated from the surrogate model: its linear coefficients represent the importance of the tokens. This model is trained with synthetic data, generated via a two-step approach where the values of the records to explain are properly altered (in the perturbation phase) and then passed to the original model to get their class (in the reconstruction phase).

Example 1.1. Figure 1 shows an example of a record describing a pair of entities. A suffix is added to the attribute names to show which entity they are describing. The application of a DL based model to the record (e.g. DeepMatcher) let us know that the entities in the record do not refer to the same real-world entity. This is evident for a human person, being clear from the attributes that the left entity is a digital camera and the right entity is a leather case. But the EM model is not able to explain the reasons for its choice. This is the task for an explainer which takes the record to explain, transforms it into tokens (we create a token for each space-separated term), generates a number of perturbations (typically performed by casually dropping tokens), passes each perturbation to the model to get the class, and exploits the so generated dataset to train a linear model. The coefficients associated to the linear model are the ones explaining the behavior of the EM model on the target record. The tokens `sony`, `lens`, `dsira200w` can be considered by an explainer as an evidence of the fact that the record is describing a non-matching entity.

The direct application of a perturbation mechanism based on token removals is not effective for the dataset used in EM. The reason is that removing random tokens is likely to affect both the

entities represented by the dataset item. The generated synthetic records may then contain *null perturbations* where the same tokens referring to the different entities are removed. Moreover, since the EM datasets are largely imbalanced, perturbations frequently lead to records belonging to the non-matching class. To solve this issue, Mojito [4] introduces the "COPY" perturbation mechanism, where attribute values describing one of the entities in a record are substituted to the corresponding attribute values of the second entity. The aim is to introduce a perturbation that increases the match probability between pairs of entities. The aim is to create records representing matching entities. But duplicating entire attribute values does not allow the approach to discriminate among the tokens that, thanks to the copy, will provide the same contribution in the explanation.

Landmark Explanation addresses these issues by introducing two main innovations. The first is the generation of two explanations for each dataset entry, each one explaining the model decision from the perspective of one of the two entities described in the record. These explanations are generated by selecting one of the entities constituting a dataset entry in turn as a landmark. The landmark is preserved from the perturbation, which is subjected to the other entity (the varying entity). The second is a mechanism for computing explanations for records belonging to non-matching classes. Before the perturbation, we inject additional tokens extracted from the landmark entity into the varying entity. The perturbation of the varying entity with injected tokens produces a set of synthetic entities. These will all be concatenated with the landmark entity to generate the synthetic EM dataset used to train the surrogate model. The idea is to contrast the asymmetric nature of the problem: an explanation of a matching pair is always composed of "interesting" tokens since they express the reason why the entities have been considered as matching. The same does not happen for non-matching entities, since non-matching entities have many reasons to be different. So it is difficult to generate an explanation with interesting tokens for non-matching pairs. Therefore the problem is to generate the most interesting explanations. These are the ones involving tokens from one entity that if used to describe the second entity would have brought the EM model to classify the record as matching. Thanks to the injection described above non-matching pairs are pushed to be match and the resulting explanation will be more interesting.

Example 1.2. To explain the inference of an EM model applied to the record in Figure 1 , Landmark Explanation generates two explanations. The top 3 tokens generated by the explanation with the left entity as a landmark are *leather*, *nikon* and *5811*. These tokens are the ones that best differentiate entities. This means that if the left entity were described by these tokens, the record would probably be classified in the matching class by the EM model. The top 3 tokens generated by the second explanation with the right entity as a landmark are *dsira200w*, *lens* and *849.99*.

We evaluate Landmark Explanation coupled with LIME. The results of the experiments show that the explanations generated outperform the ones of the competing approaches in accuracy and "interest" for the users.

Summarizing, the main contributions of this paper are: (1) the introduction of Landmark Explanation, a tool that extends the capability of a generic post-hoc perturbation-based explainer to generate accurate local explanations of EM models; (2) the realization of an extensive experimentation of Landmark Explanation coupled with the LIME explainer [14] to demonstrate the effectiveness and the quality of the approach in comparison with different competitor systems.

The rest of the paper is organized as follows. Section 2 introduces some related work. Section 3 introduces our approach that is evaluated in Section 4. Finally, in Section 5 we sketch out conclusion and future work.

2 RELATED WORK

Explaining AI. The interpretation of machine learning techniques represents a hot topic and two main approaches for its resolution can be identified [5]. On the one hand, there are intrinsically interpretable models, such as decision trees, rule-based and linear models, which rely on structures that can be directly interpreted by humans. On the other hand, there are techniques that analyze the behavior of black-box machine learning methods via a second intermediate model built from the first. These post-hoc interpretation methods are model-agnostic (i.e. they are applicable to any ML / DL model), however they provide less faithful explanations than intrinsically interpretable models.

Regardless of the explanation technique adopted, it is further possible to distinguish between global and local interpretations [5]. In the first case, the entire functioning of an ML / DL model is examined, while in the second its behavior is studied only locally (i.e. by explaining its logic on individual predictions).

The main exponent of the category of local post-hoc interpretation techniques is LIME [14], which exploits an interpretable linear surrogate model (e.g. Lasso) to evaluate the behavior of the original model in the neighborhood of a specific data instance. It will be used in our experiments, and an extension of it is Anchor [15], which generates explanations based on if-then rules. Some examples of global explanation systems are BRL [9] and Skater[2]. Similar techniques are *permutation feature importance* and *drop-column importance* [1], which can be used to detect the global relevance of features in any model.

In this paper we focus exclusively on local post-hoc interpretation techniques (for simplicity in the rest of the paper they will also be identified as generic "explanation systems") and we propose their adaptation, through Landmark Explanation, to the Entity Matching problem.

Explainable Entity Matching. Entity matching, that is the task that identifies the records that refer to the same real-world entity in multiple datasets, represents one of the main steps of data integration and has been under study for several years. Many techniques have been proposed: from the more traditional rule-based approaches to the most modern machine learning and deep learning methods. Some examples of the first category are [16, 18]. They are intrinsically interpretable, however, the identification of the most effective set of matching rules is a complex and non-trivial task [13].

Recently, several approaches based on Deep Learning have proved particularly effective in solving this task. Some examples are DeepER [7], DeepMatcher [12], DITTO [10] and many others [2, 19]. In addition to requiring a significant amount of annotated data and a complex configuration, the main problem with these systems is the inability to interpret their behavior, affecting their usability in business environments [3].

This motivated the realization of several studies on the use of interpretation techniques in the entity matching area [11, 17], and

[2]https://github.com/oracle/Skater

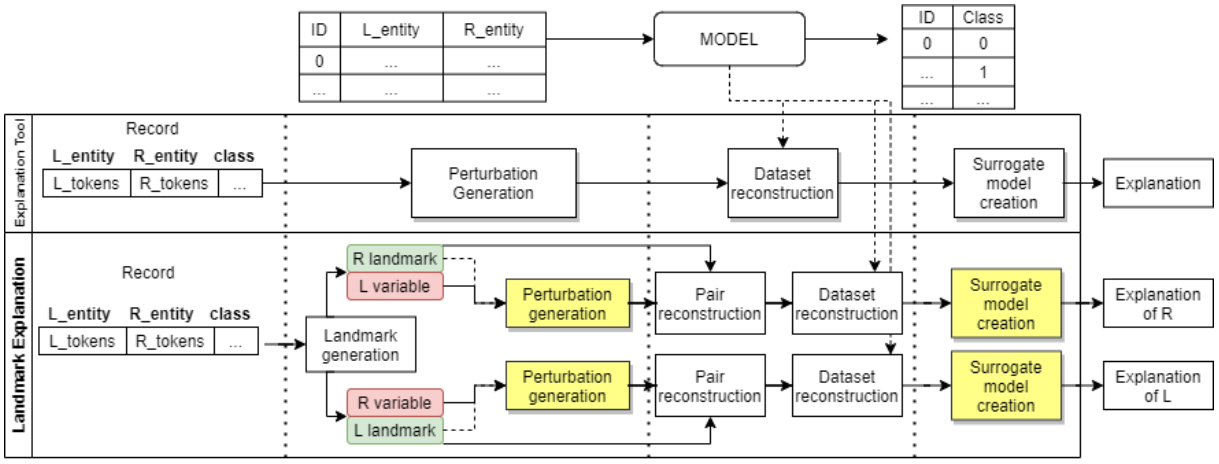


Figure 2: A generic post-hoc perturbation based explanation system (at the top of the image) compared with its extension with the Landmark Explanation **Framework (at the bottom).**

tools, like Mojito [4] and Explainer [6], have been proposed. ExplainER provides a unified interface for applying well-known interpretation techniques (e.g., LIME, Shapley, Anchor, and Skater) in the EM scenario. Mojito adapts LIME for the explanation of single EM predictions and represents the work closer to our approach. It extends LIME in two ways: 1) it exploits the subdivision of EM data into attributes, 2) it introduces a new form of data perturbation, called LIME-COPY[3], which allows generating match elements starting from non-match elements. Unlike Landmark Explanation, Mojito treats attributes atomically, distributing its impact equally to its constituent tokens. Furthermore, Landmark Explanation analyzes the diversified impact that the same token can generate depending on the entity considered as a landmark for the explanation.

3 THE LANDMARK EXPLANATION APPROACH

Landmark Explanation is a generic and extensible framework that can extend a generic local post-hoc and model-agnostic perturbation based explanation systems to the interpretation of EM model predictions. The main assumption of these generic systems is that the prediction of a model computed on a given instance can be approximated by a linear function of the predictions calculated in the neighborhood of that input instance. Their functional architecture is shown at the top of Figure 2 and can be schematized in 3 main blocks: the component for the *Perturbation generation*, for the *Dataset reconstruction* and for the *Surrogate Model Creation*.

The *Perturbation generation* component takes the record to analyze as input and generates a number of perturbations from it. These perturbations constitute a new features space, defined in the neighborhood of the record, which, once integrated by the *Dataset reconstruction* component with the predictions of the original model, can be used by the *Surrogate Model creation* component to infer its behavior in the locality of the record.

Landmark Explanation extends the generic explanation system to improve its effectiveness on datasets representing EMs. In particular, as shown at the bottom, of Figure 2, Landmark Explanation adds the *Landmark generation* component before the perturbation. This component generates for the record to explain two representations, where only one of the two entities composing the item will be subject to the perturbation and the other one will be kept fixed as a landmark. This constitutes the input for the *Perturbation generation* component that will be called twice, once for each representation. In this way, each perturbation obtained with this process will involve the attributes of one entity. The tokens of the second entity (the landmark entity) will be added by the *Pair reconstruction* component before the *Dataset reconstruction*. This allows Landmark Explanation to perturb the information of one entity at a time while preserving the pairwise structure of the EM data. A perturbation of the input entity pair is generated by varying only one of the two entities while preserving the pairwise structure of the EM data.

Note that the component performs differently when the record to explain is referring to a non-matching class. In this case, the tokens of both entities are concatenated into the varying entity and passed to the *Perturbation generation* component. The behavior of the *Pair reconstruction* component does not change, by concatenating after the perturbation the contribution of the landmark entity. This mechanism has been implemented to contrast the dataset imbalance and to generate explanations that can be more interesting for the users since based on a richer set of tokens. Finally, as for the generic explanation system, the synthetic dataset just created is used to train a linear model whose coefficients constitutes the explanation. These coefficients can be positive or negative thus indicating which tokens should be added (the one with a positive score) and which should be removed (the one with negative score) to create a description that is closed to the reference entity.

The yellow-shadowed components in the Figure are the ones provided by the explanation system we are extending. In our experiments, these components are provided by the LIME explainer. Since Landmark Explanation conceives them as black box modules, other explanations systems can be easily coupled with our approach.

3.1 Landmark Explanation components

Landmark Explanation is composed of three main components for the *Landmark generation*, the *Pairs reconstruction* and the *Dataset reconstruction*.

Landmark generation component. The goal of this component is to generate input for the *Perturbation generation* component. A

[3]In Section 4 we refer to this technique as Mojito Copy to emphasize that this technique is part of the Mojito tool.

Dataset	Type	Datasets	Size	% Match
S-BR	Structured	BeerAdvo-RateBeer	450	15.11
S-IA		iTunes-Amazon	539	24.49
S-FZ		Fodors-Zagats	946	11.63
S-DA		DBLP-ACM	12,363	17.96
S-DG		DBLP-GoogleScholar	28,707	18.63
S-AG		Amazon-Google	11,460	10.18
S-WA		Walmart-Amazon	10,242	9.39
T-AB	Textual	Abt-Buy	9,575	10.74
D-IA	Dirty	iTunes-Amazon	539	24.49
D-DA		DBLP-ACM	12,363	17.96
D-DG		DBLP-GoogleScholar	28,707	18.63
D-WA		Walmart-Amazon	10,242	9.39

Table 1: Magellan Benchmark

Tokenizer is firstly needed to transform the dataset entry in a format suitable for generating meaningful explanation. We implement a tokenization mechanism similar to the one adopted in other systems as Mojito [4] that preserves the structure of the pair of entities described in the record. A token is generated for each space-separated term in the attribute values. A prefix is introduced to each token to indicate the attribute where the original value is located in the entity schema. The prefix enumerates the tokens, to manage multiple occurrences of the same word in an attribute value.

After the tokenization, Landmark Explanation implements two mechanisms for performing this task. With the *single-entity generation*, the tokens composing the entities are separated and the perturbation component is called twice, each time with the element of a different entity. The output for each execution are the tokens of one entity (the landmark) and a number of perturbations for the second entity. This technique generates a perturbation that highlights the differences of one entity with respect to the other. It is then particularly effective when the record to explain is belonging to the matching class.

With the *double-entity generation*, the perturbation component receives as input the tokens of an artificial entity created by concatenating, for each attribute, the tokens of both the entities. The output for each execution are then the tokens of one entity (the landmark) and a number of perturbations of the artificial entity created by the concatenation. The idea of this technique is to generate the perturbation of a more extensive set of tokens (obtained by the union of the tokens of both the entities) that is effective for generating explanations for records classified as non-matching items.

Pair reconstruction component. The component receives as input the landmark entity and a number of perturbations of the tokens of the varying entity and "reconstructs" the corresponding pairs of entities (one for each perturbation). The prefixes introduced by the *Tokenizer* are exploited for this purpose and removed from the generated records.

Dataset reconstruction component. This component generates the synthetic dataset to be used for training the surrogate linear model. This is obtained by passing each pair of entities reconstructed by the previous component to the original EM model for getting the predicted class.

4 EXPERIMENTAL EVALUATION

We evaluated the explanations generated by Landmark Explanation according to two main perspectives: their reliability in representing the EM Model (in Section 4.2) and the "quality" of the explanation provided (in Section 4.3).

(a) Matching label.

	Single		Double		LIME	
	Accuracy	MAE	Accuracy	MAE	Accuracy	MAE
S-BR	**0.923**	**0.121**	0.796	0.136	0.830	0.147
S-IA	**0.940**	**0.226**	0.793	0.251	0.847	0.240
S-FZ	**0.934**	**0.228**	0.841	0.237	0.865	0.236
S-DA	0.887	0.171	**0.894**	**0.164**	0.573	0.337
S-DG	**0.836**	**0.196**	0.823	**0.196**	0.757	0.200
S-AG	0.896	**0.074**	**0.903**	0.112	0.698	0.148
S-WA	**0.954**	**0.071**	0.928	0.115	0.659	0.228
T-AB	**0.908**	**0.066**	0.854	0.146	0.758	0.118
D-IA	0.899	**0.090**	**0.975**	0.112	0.780	0.156
D-DA	0.942	**0.030**	**0.979**	0.041	0.940	**0.025**
D-D	0.929	**0.107**	**0.963**	0.152	0.891	0.115
D-WA	**0.916**	**0.045**	0.901	0.090	0.813	0.074

(b) Non-matching label.

	Single		Double		LIME		Mojito Copy	
	Accuracy	MAE	Accuracy	MAE	Accuracy	MAE	Accuracy	MAE
S-BR	0.747	0.092	**0.927**	**0.037**	0.843	0.100	0.011	0.369
S-IA	0.669	0.248	**0.736**	**0.127**	0.624	0.267	0.022	0.569
S-FZ	0.811	0.188	0.853	**0.134**	**0.953**	0.189	0.032	0.681
S-DA	0.975	**0.021**	0.590	0.287	**0.985**	0.066	0.005	0.574
S-DG	0.895	**0.086**	0.660	0.306	**0.935**	0.107	0.005	0.504
S-AG	0.835	0.107	0.895	**0.056**	**0.905**	0.097	0.010	0.445
S-WA	**0.990**	**0.028**	0.955	0.217	0.890	0.352	0.000	0.746
T-AB	**0.860**	0.076	0.680	**0.047**	0.795	0.092	0.045	0.328
D-IA	**0.874**	**0.019**	0.291	0.070	0.390	0.129	0.242	0.191
D-DA	0.615	0.071	0.300	**0.027**	**0.690**	0.036	0.010	0.173
D-DG	0.540	0.305	0.375	**0.118**	**0.640**	0.235	0.040	0.437
D-WA	0.500	0.184	**0.785**	**0.078**	0.500	0.192	0.005	0.380

Table 2: Token-based evaluation.

4.1 Experimental setup

We run the experiments on a VM deployed on Google Cloud with 12 GB of RAM, GPU K80, and Intel(R) Xeon(R) CPU @ 2.30GHz.

Dataset and Model. The EM model explained in the experiments is a Logistic Regression Classifier. We experimented Landmark Explanation against the datasets provided by the Magellan library[4] which is considered as a standard benchmark for the evaluation of EM tasks. The datasets are listed in Table1, where the size and the percentage of records representing matching entities are shown. The records in all datasets represent pairs of entities described with the same attributes. A label is provided to express if the record represents a matching / non-matching pair of entities. In the experiments, we sampled 100 records per label and we computed their explanations. Note that all records are sampled when the dataset contains less than 100 records (see for example the dataset S-BR which contains only 68 records labeled as matching entity).

4.2 Reliability of the explanations

The goal of the experiment is to evaluate the reliability of the explanations generated by Landmark Explanation in interpreting the behavior of an EM model through single predictions. An explanation is considered reliable if it is able to consistently recognize the importance of the features with the EM model. To evaluate this, we performed two kinds of experiments, one analyzing the weights assigned by Landmark Explanation to the *tokens* it generates, the second the weights assigned by the EM model to the dataset *attributes*.

4.2.1 Token-based evaluation. Through this first kind of experiment, we evaluate if the weights assigned by Landmark Explanation to the tokens generate a surrogate model consistent with the EM model. We performed an experiment that is similar

[4]https://github.com/anhaidgroup/deepmatcher/blob/master/Datasets.md

to the one proposed in the evaluation of LIME: 25% of tokens are randomly selected and removed from the record to explain, defining a new item. We then compared the probability score obtained passing the new item to the EM model with the one of the original record, where we have subtracted the sum of the coefficients associated with the removed tokens. If the explanation model correctly represents the EM model these two values should be close. We repeated the experiment 100 times for each class (see the beginning of Section 4), and we measured the performance obtained by means of two metrics: the mean average error (MAE) and the accuracy on the predicted class. We performed the experiments for all datasets and testing all techniques for generating the perturbations as reported in Table 2. Note that column LIME shows the results obtained with LIME / Mojito Drop[5] with the same setting. Non-matching settings also include a comparison with the Mojito Copy technique, which has been designed for this kind of record.

Discussion. Table 2a shows that Landmark Explanation, applied to records labeled as matching entity, performs better than LIME in the datasets when the perturbation is generated with the single-entity technique (it obtains better accuracy in all datasets and low MAE in 11/12 datasets). The double-entity generation technique performs slightly worse: in 9/12 it obtains better accuracy and in 6/12 lower MAE). Nevertheless, the scores, when worst, are very close to LIME. Note that in some datasets there is some small contradiction between accuracy and MAE scores computed for the same dataset. For example, we observe that Landmark Explanation applied to the S-BR dataset with the double-generation configuration has a better MAE score than LIME/Mojito Drop. This is not the same for the accuracy value, where LIME performs better. This is motivated by the fact that the probability scores generated by the model are close to the decision threshold (fixed to 0.5). Then, small fluctuations in the surrogate model can generate mismatches in the class predicted by the explained model for the records, even if the EM model and the surrogate model are very close. Table 2b shows the accuracy and the MAE obtained analyzing records referring to non-matching labels. In this scenario, the double entity perturbation obtains the best scores with an accuracy better than LIME/Mojito Drop in 4/12 datasets and a lower MAE in 10/12 datasets. The reason is due to the effect of the duplicated tokens inserted in the dataset used for training the surrogate model. These tokens, being similar to the ones of the entities described in the record, push the EM model to classify the record towards a matching label even in the case of an imbalanced dataset. By using the same tokens, the entities will likely be considered by the model as similar. Conversely, the perturbations generated by LIME / Mojito drop are subsets of the original record, which, in this case, was classified as non-match. By removing tokens from descriptions of entities classified as non-matching, the probability score of the EM model usually decreases and it is unlikely to obtain descriptions of entities that a classifier evaluates as matching. If we pushed the decision threshold to 0.4 (instead of 0.5), Landmark Explanation would obtain a better performance than LIME/Mojito drop in 10/12 datasets.

Note that the copying technique introduced by Mojito to manage records associated with non-matching labels does not show high performance. The reason is that Mojito generates a perturbation by duplicating entire attributes.

The result of this operation is that the tokens of the replaced attribute have the same weights, thus decreasing the performance.

[5]the Mojito Drop technique implements the LIME approach

(a) Matching label.

	Single	Double	LIME
S-BR	1.000	1.000	1.000
S-IA	0.261	**0.538**	0.495
S-FZ	**0.592**	**0.592**	0.143
S-DA	**0.520**	**0.520**	0.200
S-DG	1.000	1.000	1.000
S-AG	**1.000**	0.545	0.545
S-WA	0.901	**1.000**	0.544
T-AB	0.545	0.545	0.545
D-IA	0.892	**0.939**	0.848
D-DA	1.000	1.000	1.000
D-DG	1.000	1.000	1.000
D-WA	0.526	**0.681**	0.526

(b) Non-matching label.

	Single	Double	LIME	Mojito Copy
S-BR	0.733	**1.000**	**1.000**	**1.000**
S-IA	0.312	0.538	0.687	**0.756**
S-FZ	0.333	0.518	**0.864**	0.414
S-DA	0.200	**1.000**	0.200	0.520
S-DG	**1.000**	0.520	**1.000**	0.333
S-AG	**1.000**	0.545	0.545	0.545
S-WA	0.573	**1.000**	0.872	**1.000**
T-AB	0.545	0.545	0.545	**1.000**
D-IA	0.925	0.899	0.776	**0.939**
D-DA	0.813	**1.000**	**1.000**	**1.000**
D-DG	**1.000**	**1.000**	**1.000**	0.813
D-WA	0.681	0.681	0.681	0.681

Table 3: Attribute-based evaluation (weighted Kendall measure applied on the ranked list of attributed as generated by the EM and the surrogate model).

Lesson learned. The surrogate model built by Landmark Explanation with the single-entity perturbation is an accurate representation of the EM model for records representing matching pairs of entities. The model built with the double-entity perturbation is an accurate representation of the EM model for record representing non-matching pairs of entities.

4.2.2 Attribute-based evaluation. The attribute-based evaluation proceeds in the opposite direction: it starts from the internal structure of the EM model and evaluates if the weights it gives to the attributes are close to the ones we can derive from the tokens obtained by Landmark Explanation. For this reason, we have analyzed the weights given to the dataset attributes by the Logistic Regression model used as EM model in the experiments and ranked the attributes according to their absolute values. We have done a similar operation with the surrogate model, where the weights of the attributes have been computed by summing the absolute weights of their composing tokens. The idea is that the order of the attributes computed on the basis of their weights should be the same in both models. In Table 3, we measured the correlation computed by applying the weighted Kendall tau correlation measure, between the ranked list of attributes of the EM and surrogate model.

Discussion. Table 3a shows the experiments on the records representing matching entity pairs. The correlation scores achieved by Landmark Explanation with the double-entity perturbation approach are better or equal to the ones achieved by LIME/Mojito for all dataset. Table 3b shows the experiments on the records representing non-matching entity pairs. In this case, the single-entity configuration obtained better/equal results than LIME/Mojito Drop in 7/12 datasets (4/12 against Mojito Copy); the double-entity configuration obtained better/equal results than LIME/Mojito drop in 9/12 datasets (the same against Mojito Copy). Note that Mojito Copy, that has been explicitly designed for non-matching entities, performs better than LIME/Mojito Drop in 5/12 datasets only and equal/close to LIME/Mojito Drop in 4/12 datasets and worst in the remaining 3 datasets.

Lesson learned. Landmark Explanation creates surrogate models that maintain a relative importance of the attributes similar to the ones of the EM model to explain.

4.3 Quality of the explanations

To introduce this experiment, let us consider an application that aims to provide the explanation for a record labeled as a non-matching entity. The tokens of non-matching are "less polarized":

(a) Matching label.

	Single	Double	LIME
S-BR	0.643	0.593	**0.686**
S-IA	0.652	0.404	**0.702**
S-FZ	0.606	0.447	**0.612**
S-DA	**1.000**	0.940	0.965
S-DG	0.660	0.610	**0.925**
S-AG	0.955	0.800	**0.990**
S-WA	**1.000**	0.785	0.870
T-AB	0.985	0.575	**0.995**
D-IA	**0.561**	0.278	0.311
D-DA	0.695	0.715	**0.800**
D-DG	0.635	0.530	**0.735**
D-WA	**0.915**	0.545	0.880

(b) Non-matching label.

	Single	Double	LIME	Mojito Copy
S-BR	0.298	**0.927**	0.331	0.011
S-IA	0.545	**0.736**	0.393	0.000
S-FZ	0.079	**0.853**	0.047	0.000
S-DA	0.000	0.030	0.000	0.005
S-DG	0.020	**0.545**	0.020	0.000
S-AG	0.075	**0.895**	0.070	0.010
S-WA	0.015	**0.955**	0.000	0.000
T-AB	0.305	**0.680**	0.340	0.045
D-IA	**0.670**	0.291	0.379	0.027
D-DA	0.205	**0.300**	0.125	0.000
D-DG	0.200	**0.375**	0.160	0.030
D-WA	0.190	**0.785**	0.130	0.005

Table 4: Evaluation of the interest associated to the computed explanations.

there are many reasons to be dissimilar for two entities. For this reason, it is easy for this application to say why two entities do not match, since there is plenty of tokens that do not match between the entities in the record. Nevertheless, the explanation would be more interesting if the tokens returned would be the ones changing class of the record from non-matching to matching. In other words, we claim that an interesting explanation for non-matching entities should return the tokens that, if shared by the second entity, would make the record classified as matching.

In this section, we describe the evaluations we performed to evaluate the aforementioned situation. The experiments are similar to the first experiments described in Section 4.2, but in this case we select the tokens to remove. For sake of completeness, we performed a similar experiment with records classified as matches even if this evaluation is less meaningful. When the record is associated with a matching label, we remove all positive tokens (all tokens that contribute to the decision). The negative tokens are removed when the label represents a non-matching record. In Table 4 to evaluate the experiment we measure the *interest*, which is the accuracy computed on the records where the removal of the tokens was able to generate a change in the label.

Discussion. Table 4a shows that Landmark Explanation is good but slightly worse than LIME in terms of interest, when the records are labeled as matching class. This happens even if the surrogate model is really accurate (the MAE score is the lowest for all experiments with the single-entity configuration). The problem is that in most of the cases, even removing all tokens, the explanation created by Landmark Explanation belongs to the same class as before the token removal. Note that if we set a decision threshold to 0.4, our approach has the best results in all datasets. Table 4b shows that the explanations of non-matching entities generated by Landmark Explanation in the setting double-entity outperform the ones of Lime/Mojito Drop and Mojito Copy.

Lesson learned. Landmark Explanation generates interesting explanations, and the perturbation made with the double-entity generation technique effectively increases "the interest" of non-matching record explanations.

5 CONCLUSION

This paper introduces Landmark Explanation a tool that makes a post-hoc perturbation-based explainer able to deal with ML and DL models describing EM datasets. The approach has been experimented coupled with the LIME explainer, which is one of the most used state of the art approaches. The results show that the explanations generated by Landmark Explanation outperform the ones generated by the competing approaches in accuracy. Moreover, the explanations generated by Landmark Explanation have been experimented to be "more interesting" for the users.

Future work includes the study of techniques for summarizing the explanations to facilitate the interpretation of the EM model as an whole.

ACKNOWLEDGEMENT

This work was partially funded by SBDIO I4.0 (https://www.sbdioi40.it/), an industrial research project funded by the POR FESR Emilia Romagna 2014-2020 as part of the Smart Specialization Strategy (S3).

REFERENCES

[1] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[2] Ursin Brunner and Kurt Stockinger. 2020. Entity Matching with Transformer Architectures - A Step Forward in Data Integration. In *EDBT*. OpenProceedings.org, 463–473.
[3] Zhaoqiang Chen, Qun Chen, Boyi Hou, Zhanhuai Li, and Guoliang Li. 2020. Towards Interpretable and Learnable Risk Analysis for Entity Resolution. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1165–1180.
[4] Vincenzo Di Cicco, Donatella Firmani, Nick Koudas, Paolo Merialdo, and Divesh Srivastava. 2019. Interpreting deep learning models for entity resolution: an experience report using LIME. In *aiDM@SIGMOD*. ACM, 8:1–8:4.
[5] Mengnan Du, Ninghao Liu, and Xia Hu. 2020. Techniques for interpretable machine learning. *Commun. ACM* 63, 1 (2020), 68–77.
[6] Amr Ebaid, Saravanan Thirumuruganathan, Walid G Aref, Ahmed Elmagarmid, and Mourad Ouzzani. 2019. Explainer: Entity resolution explanations. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2000–2003.
[7] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq R. Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed Representations of Tuples for Entity Resolution. *Proc. VLDB Endow.* 11, 11 (2018), 1454–1467.
[8] Amirata Ghorbani and James Y. Zou. 2019. Data Shapley: Equitable Valuation of Data for Machine Learning. In *ICML (Proceedings of Machine Learning Research)*, Vol. 97. PMLR, 2242–2251.
[9] Benjamin Letham, Cynthia Rudin, Tyler H McCormick, David Madigan, et al. 2015. Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model. *The Annals of Applied Statistics* 9, 3 (2015), 1350–1371.
[10] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep Entity Matching with Pre-Trained Language Models. *Proc. VLDB Endow.* 14, 1 (Sept. 2020), 50–60. https://doi.org/10.14778/3421424.3421431
[11] Xiaolan Wang Laura Haas Alexandra Meliou. 2018. Explaining Data Integration. *Data Engineering* (2018), 47.
[12] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD Conference*. ACM, 19–34.
[13] Matteo Paganelli, Paolo Sottovia, Francesco Guerra, and Yannis Velegrakis. 2019. TuneR: Fine Tuning of Rule-based Entity Matchers. In *CIKM*. ACM, 2945–2948.
[14] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. " Why should I trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.
[15] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2018. Anchors: High-Precision Model-Agnostic Explanations. In *AAAI*. AAAI Press, 1527–1535.
[16] Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. 2017. Synthesizing Entity Matching Rules by Examples. *Proc. VLDB Endow.* 11, 2 (2017), 189–202.
[17] Saravanan Thirumuruganathan, Mourad Ouzzani, and Nan Tang. 2019. Explaining Entity Resolution Predictions: Where are we and What needs to be done?. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. 1–6.
[18] Jiannan Wang, Guoliang Li, Jeffrey Xu Yu, and Jianhua Feng. 2011. Entity Matching: How Similar Is Similar. *PVLDB* (2011).
[19] Chen Zhao and Yeye He. 2019. Auto-EM: End-to-end Fuzzy Entity-Matching using Pre-trained Deep Models and Transfer Learning. In *WWW*. ACM, 2413–2424.

Short Paper

Human-Interpretable Rules for Anomaly Detection in Time-series

Ines Ben Kraiem
University of Toulouse - UT2J, IRIT
Toulouse, France
ines.ben-kraiem@irit.fr

Faiza Ghozzi
University of Sfax- ISIMS, MIRACL
Sfax, Tunisie
faiza.ghozzi@isims.usf.tn

Andre Peninou
University of Toulouse- UT2J, IRIT
Toulouse, France
andre.peninou@irit.fr

Geoffrey Roman-Jimenez
CNRS, IRIT
Toulouse, France
geoffrey.roman-jimenez@irit.fr

Olivier Teste
University of Toulouse- UT2J, IRIT
Toulouse, France
olivier.teste@irit.fr

ABSTRACT

Anomaly detection in time series is a widely studied issue in many areas. Anomalies can be detected using rule-based approaches and human-interpretable rules for anomaly detection refer to rules presented in a format that is intelligible to analysts. Learning these rules is a challenge but only a few works address the issue of detecting different types of anomalies in time-series. This paper presents an extended decision tree based on patterns to generate a minimized set of human comprehensible rules for anomaly detection in univariate times-series. This method uses Bayesian optimization to avoid manual tuning of hyper-parameters. We define a quality measure to evaluate both the accuracy and the intelligibility of the produced rules. Experiments show that our approach generates rules that outperforms the state of- the-art anomaly detection techniques.

1 INTRODUCTION

Anomaly detection in time series is a widely studied issue in many areas such as financial markets, sensor networks, habitat monitoring, network intrusion, web traffic [1], and many others. Time series are often affected by unusual events or untimely changes (e.g., measurement error or faulty sensors) that need to be detected and processed by users for analysis and exploration [7].

In a real context, experts may observe some interesting local phenomena, which can be seen as remarkable points in time series. Using their domain knowledge, the experts investigate sequences of remarkable points to detect and locate anomalies. Experts can also build decision rules manually to detect future occurrences of these anomalies. However, as the amount of collected data is increasing, the decision rules become more complex to define which makes the analysis more difficult. Automatic rule extraction and detection of different types of anomalies can be of considerable interest to an expert, leading to appropriate action that can save a lot of time and value. To overcome this challenge, rule learning algorithms have been proposed [2, 12]. Deploying such systems might reveal comprehensible information to the users to explain the root cause of anomalies better than black-box algorithms.

To address these challenges, we provide a machine learning method to generate human-interpretable rules for anomaly detection in time-series, called Composition-based Decision Tree (CDT) [1]. This method uses patterns to identify remarkable points. The compositions of remarkable points existing into time-series are learned through a specific decision tree that finally produces intelligible rules. To avoid manual tuning, we use Bayesian hyper-parameter optimization to get the best hyper-parameters for our model. The approach aims at finding the best compromise between a high accuracy during anomaly detection and a minimized set of easily human-interpretable rules. We conduct experiments on three real-world datasets and three synthetic datasets. The results show the effectiveness of our method compared to both pattern-based methods and rule-based methods for anomaly detection.

2 RELATED WORK

Numerous works on anomaly detection in time-series had been covered under various surveys and reviews [3, 13]. Several fields of study are related to pattern-based time-series data mining and rule-learning methods for anomaly detection. Pattern-based methods aim to discover frequent [4, 5] or infrequent sub-sequences [16] from a time-series. In contrast to our method, these methods are less suitable for detecting multiple anomalies and can only find a specific type of anomaly. Rule-learning methods aim to find regularities in data that can be expressed in the form of IF-THEN-like rules [2, 9, 11]. In general, the rules are evaluated based on accuracy or the number of rules produced by these algorithms. In this paper, instead of only evaluating the rules based on these criteria, we introduce a quality measure, which takes into account the number of used patterns as well as the length of rules, to make the rules simpler to interpret. We also propose a function that seeks a compromise between the interpretability of the rules and their precision.

3 METHODOLOGY

In this section, we describe our Composition-based Decision Tree (CDT) method for anomaly detection and rule extraction.

3.1 Time-Series Preprocessing

Definition 1. An univariate *time-series* is defined as $Ts = \{x_1, ..., x_n\}$ where $\forall i \in [1..n], x_i \in \mathbb{R}$ such that values x_i are uniformly spaced in time and n is the size of Ts.

The time series are collected from different sensors and the values of measures are on different ranges. To achieve scale and offset invariance, we normalize each continuous time-series Ts to values within the range $[0, 1]$. Resampling could also be used to provide additional structure or to smooth time series and remove

[1] https://github.com/IBK-TLS/CDT

any noise; e.g., downsampling reduces the frequency of time-series observations.

3.2 Time-Series Labeling

To detect anomalies, experts first analyze the neighborhood of a point (unusual variations) such as the point which precedes it and follows it to decide if is a remarkable point. Based on this idea, we label each point of the time series by checking every three successive points using patterns.

Let us considers three successive points such as x_{i-1}, x_i, x_{i+1} of a time-series. Considering all possible variations between these three points, we define nine possible variations, which can occur between successive points, as listed in Table 1 namely, PP (Positive Peak), PN (Negative Peak), SCP (Start Constant Positive), SCN (Start Constant Negative), ECP (End Constant Positive), ECN (End Constant Negative), CST (Constant), VP (Variation Positive) and VN (Variation Negative). Each of these variations can have different magnitudes into [-1,1]. To identify fine variations of values between three points, we refine each variation by defining intervals of variations into [-1,1].

Hyper-parameter (δ). We denote δ the hyper-parameter used to distinguish the different magnitudes considered for each of the nine variations (PP, PN, SCP, SCN, ECP ECN, CP, VN, and CST). δ allows the introduction of fine amplitude shifts in the variations to capture the shape complexity in time-series, typically small or large amplitudes. δ represents the number of disjointed sub-intervals considered in [-1,1] and will be determined automatically using Bayesian optimization. For a given δ, we construct $2\delta+1$ intervals : i) δ sub-intervals for positive variations in $]0,1]$, ii) δ intervals for negative variations in $[-1,0[$, and iii) 1 special case for the absence of variation (equal to 0).

For the sake of simplicity, in the rest of the paper, we only consider notation with $\delta = 2$ (other values of δ will only result in a larger variety of intervals and patterns), and we denote the 5 resulting intervals as follows: Low (L) =]0,0.5], High (H) =]0.5,1], -Low (-L) = [-0.5,0[, -High (-H) = [-1,-0.5[, and the special case, Zero (Z) = 0.

Definition 2. We define a *pattern* annotated $P = (l, \alpha, \beta)$ where l is a name (or label) identifying the pattern, and α and β are two possible intervals from [-1,1]. For each successive points x_{i-1}, x_i, x_{i+1}, the point x_i is checked by a pattern only if $x_i - x_{i-1} \in \alpha \wedge x_i - x_{i+1} \in \beta$. In this case, x_i is labeled with l. For the rest of the paper, labels are denoted by the name of the variation (PP, PN, etc.).

Fig. 1 (left) illustrates an example of a pattern $PP_{L,H}$ that helps us to find one remarkable point from a time-series. This time-series represents the consumption data of a building's calorie sensor. Fig. 1 (right) shows examples of different magnitudes of pattern such as $PP_{L,H}$, $PP_{L,L}$ and $PP_{H,H}$. Using these patterns, we can automatically label each point in time series.

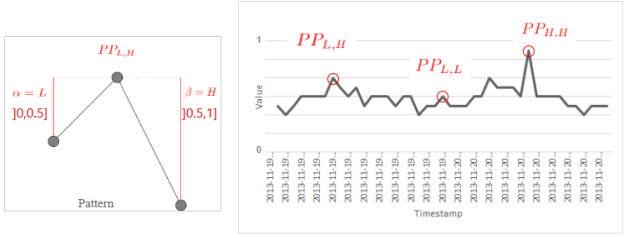


Figure 1: Example of different magnitudes of a pattern.

Definition 3. A *labeled times-series* annotated $Tsb = \{l_1, ..., l_N\}$ with $N = n - 2$ where each x_i point of the initial time-series (Ts) is replaced by the label of its corresponding pattern.

Table 1: Types of variation for labelling.

Variation	Definition	Pattern	Example
PP	$x_{i-1} < x_i \wedge x_i > x_{i+1}$		
	$\alpha, \beta \in \{L, H\}$	$PP_{\alpha,\beta}$	$PP_{L,H}$
PN	$x_{i-1} > x_i \wedge x_i < x_{i+1}$		
	$\alpha, \beta \in \{-L, -H\}$	$PN_{\alpha,\beta}$	$PN_{-H,-H}$
SCN	$x_{i-1} > x_i \wedge x_i = x_{i+1}$		
	$\beta \in \{Z\}$ $\alpha \in \{-L, -H\}$	$SCN_{\alpha,\beta}$	$SCN_{-L,0}$
SCP	$x_{i-1} < x_i \wedge x_i = x_{i+1}$		
	$\beta \in \{Z\}$ $\alpha \in \{L, H\}$	$SCP_{\alpha,\beta}$	$SCP_{H,0}$
ECN	$x_{i-1} = x_i \wedge x_i > x_{i+1}$		
	$\alpha \in \{Z\}$ $\beta \in \{L, H\}$	$ECN_{\alpha,\beta}$	$ECN_{0,L}$
ECP	$x_{i-1} = x_i \wedge x_i < x_{i+1}$		
	$\alpha \in \{Z\}$ $\beta \in \{-L, -H\}$	$ECP_{\alpha,\beta}$	$ECP_{0,-L}$
CST	$x_{i-1} = x_i \wedge x_i = x_{i+1}$		
	$\alpha, \beta \in \{Z\}$	$CST_{\alpha,\beta}$	$CST_{0,0}$
VP	$x_{i-1} < x_i \wedge x_i < x_{i+1}$		
	$\alpha \in \{L, H\}$ $\beta \in \{-L, -H\}$	$VP_{\alpha,\beta}$	$VP_{L,-L}$
VN	$x_{i-1} > x_i \wedge x_i > x_{i+1}$		
	$\alpha \in \{-L, -H\}$ $\beta \in \{L, H\}$	$VN_{\alpha,\beta}$	$VN_{-L,L}$

3.3 Composition-based Decision Tree

Given the time series labeling, we built an extension of the decision tree based on pattern compositions to produce rules able to detect anomalies.

Classically, a decision tree is induced from observations composed of feature values and a class label. It is built by splitting the training data into subsets by choosing the feature which best partitions the training data according to an evaluation criterion (e.g., Shannon's entropy, Gini index). This criterion characterizes the homogeneity of the subsets obtained by division of the data set. This process is recursively repeated on each derived subset until all instances in a subset belong to the same class label [10].

The classical decision tree considers features without any order when splitting the datasets. Conversely, in our approach, we would like to keep the order of the time series. Thus, our decision tree is built, by considering nodes as pattern compositions (ordered sequences of remarkable points) with the highest information gain. These compositions are calculated from a set of observations (sub-sequences of labeled time-series). The input of the tree is constructed by creating fixed sized sliding windows.

Definition 4. A *set of observations* annotated $D = \{d_1, d_2, ..., d_{N-\omega+1}\} = \{\{l_1, ..., l_\omega\}, \{l_2, ..., l_{\omega+1}\}, ..., \{l_{N-\omega+1}, ..., l_N\}\}$ represents the result of cutting Tsb by a sliding window of size ω, and using a fixed step size equal to one. Let M be the number of classes of observations. In our context, we consider two classes (M = 2): the abnormal class (observation with anomaly), or the normal class (observation without anomaly). Each d_i observation is associated with only one class annotated $class(d_i)$.

To determine a probability distribution of the observations over the classes, we introduce an impurity measure of a set of observations $D_j \subseteq D$. In our method, we opt to the Gini index. The *Gini impurity index*, annotated $G(D_j)$, provides a measure of the quality of D_j according to the distribution of the observations into the classes. The impurity metric is minimal (equal to 0) if a set contains only observations of one class, and it is maximal (equal to 0.5) if the set contains equally observations of all classes.

Hyper-parameter (ω). We denote $\omega <= N/2$ the window size to define observations. This hyper-parameter will be determined automatically using Bayesian optimization.

From an *observation with anomaly* we can define a composition used to split a node into two sub-nodes.

Definition 5. A *composition* annotated c is a sub-sequence of labels of an observation d_i. We denote $c \subseteq_o d_i$.

Example. Considering $d = \{l_1, l_2, l_3, l_4, l_5, l_6\}$, some compositions are $c = \{l_2, l_3, l_4\} \subseteq_o d$, $c = \{l_3, l_2, l_4\} \not\subseteq_o d$, and $c = \{l_1, l_2, l_3, l_4, l_5, l_6\} \subseteq_o d$.
We also introduce additional notations: $c \in_o D$ when $\forall d \in D, c \subseteq_o d$, and $c \notin_o D$ when $\forall d \in D, c \not\subseteq_o d$.

A decision tree is built based on features that have the highest Information Gain [10]. In CDT, the compositions are compared according to the information gain, noted IG, they provide.

The entire flow of the CDT approach we proposed is described by Algorithm 1. This algorithm builds a decision tree; we define a tree node as a quadruplet: *observations* (the set of observations considered in this node), a *composition* (used to split *observations* in two child nodes), *childTrue* (the node of observations satisfying the *composition*), and *childFalse* (the node of observations that do not satisfy the *composition*). An example corresponding to the root node is given in line 1. We introduce a function $list\_of\_all\_possible\_compositions()$ to compute all compositions deduced from a set D_j (line 6). For each composition, we calculate the information gain to split a node (line 7–15). At line 16, if $G(D_j) \neq 0$ means that the set of observations of the node is impure (observations are of different classes). Moreover, $maxGain \neq 0$ means that a composition that splits the set of observations has previously been found: in this case, we create a node N_{inc} that will determine the positive branch of the node ($c \in_o D_j$), whereas N_{exc} will be the negative one ($c \notin_o D_j$) (line 16–25). We repeat these steps until there are no more nodes to process (line 3–26).

Example. Fig. 2 illustrates an example of CDT result. The root-node is D_1, and it represents the whole training set of observations used for the construction of the CDT. The leave-nodes represent class labels and branches represent conjunctions of compositions that lead to those class labels. As shown in Fig. 2, the CDT is composed of 3 splits constructing a set of 3 leaves $\mathcal{S} = \{S_1, S_2, S_3\}$.

3.4 Rule Generation for Anomaly Detection

We convert the CDT into a set of decision rules. We only consider "pure leaf-nodes" leading to the anomaly class.

Definition 6. A *rule predicate*, annotated Rs is a branch of the decision tree leading to the anomaly class. It is constructed by combining (conjunction) the successive compositions c_i or $\neg c_i$ from the leaf-node to the root-node. For each positive branch ($c_i \in_o D_j$), the positive composition c_i is deduced whereas a negative composition $\neg c_i$ is deduced from a negative branch ($c_i \notin_o D_j$).

Algorithm 1 CDT: Composition-based Decision Tree

Input: $D = \{d_1, d_2, ..., d_{N-\omega+1}\}$ a set of observations
Output: N_{root} the root node of CDT

1: $N_{root} \leftarrow Node(D, null, null, null)$
2: $q \leftarrow [N_{root}]$ *// construct the queue of nodes to split*
3: **while** $q \neq \emptyset$ **do**
4: $N_j \leftarrow q.pop()$ *// dequeue the first node from the queue*
5: $D_j \leftarrow N_j.observations$
6: $C_j \leftarrow list\_of\_all\_possible\_compositions(D_j)$
7: $maxGain \leftarrow 0$
8: $c_{best} \leftarrow null$
9: *// Choose the composition that has the best Gain*
10: **for all** $c \in C_j$ **do**
11: **if** $IG(D_j, c) > maxGain$ **then**
12: $maxGain \leftarrow IG(D_j, c)$
13: $c_{best} \leftarrow c$
14: **end if**
15: **end for**
16: **if** $G(D_j) \neq 0$ and $maxGain \neq 0$ **then**
17: $D_{inc} \leftarrow \{d \in D_j | c_{best} \in_o d\}$
18: $D_{exc} \leftarrow \{d \in D_j | c_{best} \notin_o d\}$
19: $N_{inc} \leftarrow Node(D_{inc}, null, null, null)$
20: $N_{exc} \leftarrow Node(D_{exc}, null, null, null)$
21: $q.append(N_{inc})$ *// enqueue child nodes*
22: $q.append(N_{exc})$
23: $N_j.composition \leftarrow c_{best}$
24: $N_j.childTrue \leftarrow N_{inc}$
25: $N_j.childFalse \leftarrow N_{exc}$
26: **end if**
27: **end while**
28: **return** N_{root}

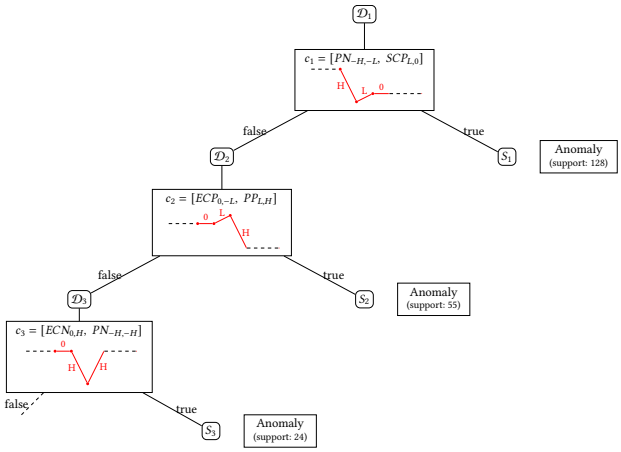


Figure 2: Illustration of a Composition-based Decision Tree (CDT).

Example. Three rules predicate are produced from the CDT in Fig. 2.

- $R_{S_1} : c_1 = [PN_{-H,-L},\ SCP_{L,0}]$
- $R_{S_2} : c_2 \wedge \neg c_1 = [ECP_{0,-L},\ PP_{L,H}] \wedge \neg[PN_{-H,-L},\ SCP_{L,0}]$
- $R_{S_3} : c_3 \wedge \neg c_2 \wedge \neg c_1 = [ECN_{0,H},\ PN_{-H,-H}] \wedge \neg[ECP_{0,-L},\ PP_{L,H}] \wedge \neg[PN_{-H,-L},\ SCP_{L,0}]$.

Using abusive notations, $c_i \wedge \neg c_j$ means that for an observation d on a time-series, we check $c_i \subseteq_o d \wedge c_j \not\subseteq_o d$.

Definition 7. A rule, annotated $\mathcal{R}$, is a disjunction of rule predicates. For instance, as shown in Fig. 2, $\mathcal{R} = R_{S_1} \vee R_{S_2} \vee R_{S_3} = (c_1) \vee (c_2 \wedge \neg c_1) \vee (c_3 \wedge \neg c_2 \wedge \neg c_1)$.

Rule Simplifications. One way to minimize CDT's rules is to post-process the produced rules through Boolean algebra simplifications. We aim to minimize the "sum-of-products" forms of Boolean functions. In our approach, this inter-branch simplification is applied until there is no longer any simplification to be made. This allows us to minimize the number of compositions in a rule. Using Boolean algebra, we can simplify the rule $\mathcal{R}$ generated from the tree in Fig. 2 as $\mathcal{R} = (c_1) \vee (c_2 \wedge \neg c_1) \vee (c_3 \wedge \neg c_2 \wedge \neg c_1) = (c_1) \vee (c_2) \vee (c_3)$.

3.5 Quality Measure

We aim to generate rules that are both accurate and comprehensible. According to experts opinion, a comprehensible rule should be short [2, 9] and should contain a minimized number of various labels. Therefore, we defined the following criteria to evaluate a quality of rules:

- $\mathcal{I}(c)$ to characterize the *quality of a composition* c depending on its length and the number of patterns used;
- $\mathcal{M}(R_S)$ to characterize the *quality of a rule predicate* R_S depending on the number of compositions and the quality of each one ($\mathcal{I}(c)$);
- $\mathcal{Q}(\mathcal{R})$ to characterize the *quality of a rule* $\mathcal{R}$ depending on the quality of rule predicate and its support.

We first calculate the interpretability of a composition as:

$$\mathcal{I}(c) = 1 - \frac{L_c.N_L}{\omega.MaxL} \tag{1}$$

where $L_c = |c|$ denotes the length of a composition c, N_L is the number of unique labels used in a composition, ω is the maximum window size, and $MaxL$ is the total number of labels. Then, we calculate the average interpretability of a rule predicate (conjunction of compositions) as:

$$\mathcal{M}(R_S) = \frac{1}{Nc} \sum_{k=1}^{Nc} \mathcal{I}(c_k) \tag{2}$$

where Nc is the number of compositions in a rule predicate R_S. Finally, extracted rules' quality is calculated as:

$$\mathcal{Q}(\mathcal{R}) = \frac{1}{S} \sum_{i=1}^{N_{Rs}} S_{R_{S_i}} . \mathcal{M}(R_{S_i}) \tag{3}$$

where N_{Rs} is the number of rule predicates in $\mathcal{R}$, $S_{R_{S_i}}$ is the support of rule predicate (true positive) and S is the support of all rules predicates (true positive and true negative). A rule predicate with high support is considered more important. Therefore, we multiplicate the average interpretability of a rule predicate with its support.

3.6 Hyper-Parameters selection

The manual tuning of hyper-parameters requires a prior knowledge. Automatic search algorithms such as grid search and random search could give good results. However, grid search is time consuming and random search might not find the optimal set. To address this problem, we use Bayesian Optimization [14] to efficiently get the best hyper-parameters (δ, ω) for our model. Indeed, we aim to find a configuration (that is, a set of parameters) that maximizes a performance metric or an objective function. Hence, we defined a search space and we try to find the hyper-parameters values of CDT that yield the highest result as measured on a validation set. Hyper-parameter optimization is presented as:

$$h^* = \arg\max_{h \in H} F(h) \tag{4}$$

where $F(h)$ represents an objective function to maximize, $h*$ is the set of optimized hyper-parameters (δ, ω) and h can take any value in search space H.

To optimize the trade-off between detection performance and good quality of rules, we defined the objective function $F(h)$ as the F-measure weighted by our $\mathcal{Q}(\mathcal{R})$ rules' quality measure.

$$F(h) = F1(h).\mathcal{Q}(\mathcal{R}) \tag{5}$$

where $F1(h)$ is the F-measure (the harmonic mean of the precision and recall) of the classification performance obtained with the set of parameters (h).

4 EXPERIMENTS

For our evaluation, we use real-world datasets from the Management and Exploitation Service (SGE) [6], and data from the Yahoo datasets [8]. We compare CDT with state-of-the-art pattern-based as well as rule-based methods for anomaly detection.

In SGE data sets, we aim to handle anomalies on calorie and electric consumption datasets. Calorie data consists of 25 consumption datasets from sensors deployed in different buildings managed by the SGE. These measurements are daily data for more than three years, about 33536 observations in total and they contain 586 anomalies. Electricity measurements are collected hourly for 10 years from one sensor (96074 in total). There are in total 10343 anomalies in the electricity dataset.

The Webscope S5 dataset, which is publicly available in [8], consists of 371 files divided into four categories, named A1/A2/A3/A4, each one containing respectively, 67/100/100/100 files. A1 Benchmark is based on real production traffic from actual web services while classes A2, A3, and A4 contain synthetic anomaly data. These datasets are represented by time-series in one-hour units. There are a total of 94778 traffic values in 67 different files and 1669 of these values are abnormal. The anomalies in the synthetic datasets are inserted at random positions. A2 Benchmark contains 142002 values with 466 anomalies while 168000 values exist in A3 and A4 Benchmarks with respectively 943 and 837 anomalies.

4.1 Evaluation Process and Metrics

The performance of all the methods is compared based upon the F1 score, the rules'quality metric $\mathcal{Q}$, and the objective function $F(h)$ used as defined in equation (5). We use the $F1$ score and $\mathcal{Q}$ score to compare the accuracy of our method against pattern-based methods. We use the $F(h)$ score to evaluate both the accuracy and the interpretability of the rules generated by our CDT method compared to those of rule learning methods. For evaluation, we split every dataset into three subsets: training set (60%), validation set (20%), and testing set (20%) ratio. We use the train and validation set to optimize the model's hyper-parameter values using Bayesian optimization. Then, we evaluate the optimized model on testing set.

Hyper-Parameters Optimization. To limit the search space of the Bayesian optimization, we constrained the parameter ω within [3,31] and δ within [1,21]. Table 2 shows the optimal hyper-parameters found with the Bayesian optimization. As we can see in Table 2, optimization on $F(h)$ tends to favors a small number of splits (δ) for patterns when compared to the $F1$ score optimization. This is due to the quality measure of rules $\mathcal{Q}(\mathcal{R})$,

which looks for short rules that include a minimum number of labels (δ). However, the size of observations (ω) needed to construct an optimal CDT remains to be comparable for $F1$ and $F(h)$ suggesting the need for presence of neighbors surrounding an anomaly to achieve good anomaly detection with CDT.

Table 2: Parameters of CDT for experiments.

Evaluation	F1-score		F(h)-score	
Dataset	ω	δ	ω	δ
SGE_Electricity	27	2	27	2
SGE_Calorie	5	4	21	1
Yahoo_A1	27	16	25	1
Yahoo_A2	17	2	17	1
Yahoo_A3	29	12	17	1
Yahoo_A4	25	8	21	1

4.2 Experiments with Pattern-based Algorithms

We employ the following three approaches as baseline methods to compare with our approach:

- Pattern-Based Anomaly Detection (PBAD) is an anomaly detection method based on frequent pattern mining techniques in mixed-type time-series [4].
- Matrix Profile (MP) is an anomaly detection method based on similarity-join to detect time series discords [15].
- Pattern Anomaly Value (PAV) is an anomaly detection algorithm based on pattern anomaly value. The anomalies are the infrequent linear pattern [16] .

To evaluate these methods, we used the implementation available in [4]. These algorithms are window-based approaches. Hence, we used the recommended settings for each of them. We split the time series into sliding windows of length 12 with a step size 6. These anomaly detection algorithms provide an anomaly score for each window. As these algorithms are unsupervised, we build the anomaly detection model on the full-time series data and we evaluate it using $F1$ score. For CDT, we used the appropriate values of hyper-parameters calculated using F1-score as provided in Table 2. All the data sets are normalized between 0 and 1 during the pre-processing phase. For Yahoo datasets and SGE-Electricity we downsampled these datasets from hours to days.

Result Analysis. Table 3 provides the $F1$ score obtained by each algorithm on each of the six univariate time-series datasets. The maximum values of $F1$ score for each dataset are given in bold type. We also calculate the average rank of each method. CDT outperforms the existing baselines in five of the six datasets. It can be observed in Table 3 that our method is more stable for different datasets than baselines. Note that for the competing algorithms, the data should be balanced otherwise, the detection results are poor. We have tested PBAD, PAV and MP on our initial datasets and on a balanced version and we have noticed that its performance highly degrades on the first case. In fact, they tend to focus on the accuracy of predictions from the majority class (normal class) which generates poor precision for the minority class (anomaly class). Therefore, the results of PBAD, PAV and MP in Table 3 are obtained on the balanced data.

Table 3: Evaluation of Anomaly Detection using F1-score for CDT and Pattern-based algorithms.

Dataset \ Algorithm	CDT	PBAD	PAV	MP
SGE_Electricity	**0.76**	0.70	0.74	0.70
SGE_Calorie	0.85	0.80	0.88	**0.91**
Yahoo_A1	**0.92**	0.72	0.75	0.76
Yahoo_A2	**0.99**	0.65	**0.99**	0.76
Yahoo_A3	**1.0**	0.73	0.99	0.70
Yahoo_A4	**0.98**	0.75	0.93	0.96
Average	**0.92**	0.72	0.88	0.80

4.3 Experiments with Rule Learning Algorithms

We compare our CDT method with the following state-of-the-art rule learning algorithms:

- PART is a combination of C4.5 and RIPPER rule learning to produce rules from partial decision trees using C4.5 algorithm [9].
- JRip implements a rule learner and incremental pruning to produce error reduction (RIPPER) [12]. Rules are formed by greedily adding conditions to the antecedent of a rule.

We compare CDT with PART and JRip based on $F1$ score, $Q(\mathcal{R})$ and $F(h)$ score (Table 4) and the number of rules produced by the classifiers (Figure 3). We evaluated these methods using WEKA. We use 10-fold cross validation to test and evaluate the PART and JRip with the standard default setting of WEKA. For CDT and each competitor, we use the hyper-parameters values obtained to maximize $F(h) - score$ as provided in Table 2.

Result Analysis. Table 4 shows the comparison results for each algorithm in the six datasets using $F1$, $Q(\mathcal{R})$ and $F(h)$ score. Note that the results of the F1 score for CDT in Table 4 are different from those in Table 3 because they are not evaluated with the same hyperparameter values (Table 2).

Overall, the average scores show that our approach has got the first position in ranking followed by PART and JRip. CDT outperforms PART and JRip in all datasets in the $F1$ score, in three of the six datasets in the $Q(\mathcal{R})$ score and all datasets in the $F(h)$ score. We can observe from the Table 4 that JRip has a high quality of rules $Q(\mathcal{R})$ in three datasets as well as CDT. This is due to the size of its generated rules that are quite short. However, it is less accurate than CDT and PART in almost all datasets. We can also see that none of the baseline algorithms has good $F(h)$ overall data sets. While CDT has the best tradeoff between $F1$ score and $Q(\mathcal{R})$ score.

Fig. 3 shows a summary of the number of rules produced by each method. CDT produces a fewer number of rules between 5 and 16 rules. It is followed by JRip that produced reasonably few rules between 15 and 30 rules. However, PART highly produces rules that are between 24 and 142. This is due to the specificity of the rules generated that have low support.

We present some examples of the generated rules by our CDT algorithm from SGE data sets to detect multiple anomalies in Table 5. As we can see, the rules with visualized patterns are easy to intuitively understand and can be easily interpreted by users. The experts give the following comments: the negative peak is considered an anomaly because the energy consumption in a building cannot be negative. The positive peak has occurred following overconsumption in the building. The collective anomalies present abnormal variations in successive points. This is

Table 4: Evaluation of anomaly detection using the $F1$ score, the quality measure $Q(\mathcal{R})$ and the objective function $F(h)$.

Evaluation	F1-score			$Q(\mathcal{R})$			F(h)-score		
Algorithm / Dataset	CDT	PART	JRip	CDT	PART	JRip	CDT	PART	JRip
SGE_Electricity	**0.76**	0.71	0.72	0.67	0.67	**0.70**	**0.51**	0.48	0.50
SGE_Calorie	**0.99**	0.80	0.79	0.61	0.65	**0.69**	**0.60**	0.52	0.54
Yahoo_A1	**0.91**	0.70	0.69	0.48	0.50	**0.56**	**0.43**	0.35	0.39
Yahoo_A2	**0.99**	0.80	0.77	**0.69**	0.68	0.65	**0.68**	0.54	0.50
Yahoo_A3	**0.98**	0.78	0.71	**0.77**	0.69	0.70	**0.75**	0.54	0.50
Yahoo_A4	**0.97**	0.73	0.75	**0.70**	**0.70**	0.68	**0.68**	0.51	0.51
Average	**0.93**	0.75	0.74	**0.65**	0.64	0.64	**0.61**	0.49	0.49

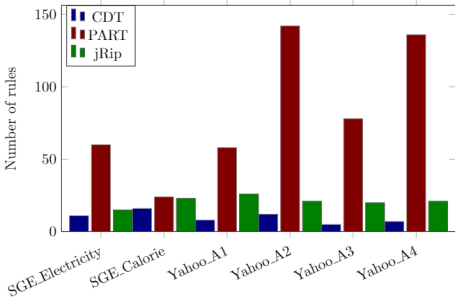


Figure 3: The number of rules generated for anomaly detection.

due to a fault in the reading of the meters. Finally, the constant anomaly illustrated a stop of the meter.

Table 5: Example of rules generated for anomaly detection in the SGE_Calorie datasets.

Rule Predicate	Representation	Type of anomaly
$[PN_{-H,-H}, SCP_{-H,0}, CST, CST, CST]$ and $\neg\,[ECN_{0,-H}, SCN_{-H,0}]$		negative peak
$[ECP_{0,-L}, PP_{L,H}]$		positive peak
$[PN_{-H,-H}, PP_{H,H}]$		collective anomaly
$[SCN_{H,0}, CST, CST, CST, CST, CST]$ and $\neg\,[PN_{-H,-H}, SCP_{H,0}, CST]$		constant anomaly

5 CONCLUSION

We propose a machine learning method, CDT, that generates human-interpretable rules based on a formalisation of 9 general patterns of variations for multiple anomaly detection in time-series. The approach is based on a modified decision tree which considers nodes as pattern compositions. Using Bayesian Optimization, we optimized the hyper-parameters such that it maximizes both the rules'quality and the classification performances. The performance of the presented method was tested using the SGE and Yahoo datasets. Our approach appeared as robust compared to the existing algorithms in conducted experiments where their model accuracy decreases in case of multiple anomalies and in generating few interpretable rules.

Future work will concern the improvement of the generated rules. For instance, combine rules by a generalization and eliminate redundant rules. Moreover, we could investigate other hyper-parameters such as the size of down-sampling to improve the quality of the generated rules. We could also expand our method to suit multivariate time-series.

Acknowledgment. This PhD. was supported by the Management and Exploitation Service (SGE) of the Rangueil campus attached to the Rectorate of Toulouse and the research is made in the context of the neOCampus project (Paul Sabatier University, Toulouse).

REFERENCES

[1] Charu C. Aggarwal. 2015. Outlier analysis. In Data mining. Springer, Cham, 237–263.
[2] Nahla Barakat and Joachim Diederich. 2005. Eclectic rule-extraction from support vector machines. International Journal of Computational Intelligence 2, 1 (2005), 59–62.
[3] Sina Däubener, Sebastian Schmitt Hao Wang, Peter Krause, and Thomas Bäck. 2019. Anomaly Detection in Univariate Time Series: An Empirical Comparison of Machine Learning Algorithms. ICDM (2019).
[4] Len Feremans, Vincent Vercruyssen, Boris Cule, Wannes Meert, , and Bart Goethals. 2019. Pattern-based anomaly detection in mixed-type time series. Lecture Notes in Artificial Intelligence (2019).
[5] Zengyou He, Xiaofei Xu, Joshua Zhexue Huang, and Shengchun Deng. 2005. FP-outlier: Frequent pattern based outlier detection. Computer Science and Information Systems 2, 1 (2005), 103–118.
[6] Ines Ben kraiem, André Péninou Faiza Ghozzi, Geoffrey Roman-Jimenez, and Olivier Teste. 2020. Automatic Classification Rules for Anomaly Detection in Time-series. RCIS (2020).
[7] Ines Ben kraiem, André Péninou Faiza Ghozzi, and Olivier Teste. 2019. Pattern-based method for anomaly detection in sensor networks. International Conference on Enterprise Information Systems 1 (jan 2019), 104–113.
[8] Nikolay Laptev and Saeed Amizadeh. 2015. A labeled anomaly detection dataset S5 Yahoo Research, v1. https://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=70
[9] Daud Nor Ridzuan and Corne David Wolfe. 2009. Human readable rule induction in medical data mining. In Proceedings of the European Computing Conference (vol.1 ed.), Vol. 27 LNEE. 787–798.
[10] Jiang Su and Harry Zhang. 2006. A fast decision tree learning algorithm. In AAAI, Vol. 6. 500–505.
[11] William W.Cohen. 1995. Fast effective rule induction. In Machine learning proceedings 1995. Elsevier, 115–123.
[12] Ian H. Witten and Eibe Frank. 2005. Data Mining: Practical Machine Learning Tools and Techniques (2nd ed.). Morgan Kaufmann.
[13] Hu-Sheng Wu. 2016. A survey of research on anomaly detection for time series. In 2016 13th International Computer Conference on Wavelet Active Media Technology and Information Processing. IEEE, 426–431.
[14] Jia Wu, Xiu-Yun Chen, Hao Zhang, and al. 2019. Hyperparameter Optimization for Machine Learning Models Based on Bayesian Optimization. Journal of Electronic Science and Technology 17 (2019), 26.
[15] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, and al. 2016. Matrix profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In 2016 IEEE 16th international conference on data mining (ICDM). 1317–1322.
[16] Xiao yun Chen and Yan yan Zhan. 2008. Multi-scale anomaly detection algorithm based on infrequent pattern of time series. J. Comput. Appl. Math. 214, 1 (2008), 227–237.

Short Paper

DBMS Performance Troubleshooting in Cloud Computing Using SQL Transaction Clustering

Arunprasad P. Marathe
Huawei Research Canada
Markham, Ontario, Canada
arun.marathe@huawei.com,ap.marathe@gmail.com

ABSTRACT

Database management systems traditionally provide user-, table-, index-, and schema-level monitoring. For cloud-deployed applications, the research reported herein provides preliminary evidence that transaction cluster-level monitoring simplifies performance troubleshooting—especially for online transaction processing (OLTP) applications. Specifically, problematic rollbacks, performance drifts, system-wide performance problems, and system bottlenecks can be more easily debugged. The DBSCAN algorithm identifies transaction clusters based on transaction features extracted directly from a DBMS server—a job previously done using SQL log-mining. DBSCAN produces more accurate clusters when inter-transaction distances are computed using the angular cosine distance function (*ACD*) rather than the usual Euclidean distance function. Choice of *ACD* also simplifies DBSCAN parameter tuning—a task known to be nontrivial.

1 INTRODUCTION

A user books air tickets for himself and family members using an online web portal—front-end to a prototypical online transaction processing (OLTP) application. He makes several flight searches, books tickets, and provides frequent flyer numbers of the passengers. The airline reservation system implements this activity using a transaction. A second user performs different flight searches before booking a ticket for herself, but does not provide a frequent flyer number because it is not handy. The two transactions have both similarities and differences. Because they access the same tables in similar fashions, there may be a reason to believe that their performances are similar—a hypothesis that can be exploited if found true.

A study of the various applications contained in two popular OLTP benchmarking toolkits OLTP-Bench [2] and Sysbench [6] reveals that each application contains transactions that can be neatly divided into a small number of non-overlapping *transaction clusters* (between 1 and 10 for the two toolkits).

Self-similar transactions within a cluster differ in parameter values, statement orders, statement counts, statement types, rows read or updated, and so on. Nevertheless, this research shows that each cluster has a characteristic performance profile—termed its *signature*—at the level of which an OLTP application can be monitored. Sample cluster-levels metrics are average values of: transactions/sec (TPS); number of rows (read, updated, or sent to client); locking time; and so on.

Cluster-level monitoring is much simpler than transaction- or statement-level monitoring, and cluster count is independent of an OLTP application's load. Benefits of clustering multiply when that OLTP application is deployed in cloud where DBA's have to monitor performance of many applications simultaneously [17]. This paper demonstrates how transaction clustering helps a cloud DBA simplify debugging of several performance problems: identification of problematic transaction rollbacks; performance bottlenecks; system-wide performance issues; performance drifts; and so on. The cluster-level performance monitoring is not meant to replace existing tools: table-level and index-level data will continue to provide the necessary drill-downs, but help in determining where to drill-down should be valuable.

The DBSCAN algorithm [3] determines transaction clusters. DBSCAN is usually run with the Euclidean distance function, but this research demonstrates that when used with a normalized distance function called the *angular cosine distance* (*ACD*), DBSCAN finds more accurate clusters, and DBSCAN parameter tuning becomes easier—welcome news for a cloud DBA who cannot hand-tune the parameters for each OLTP application. DBSCAN parameter tuning is a known difficult task [4, 15], and therefore, suitability of *ACD* is a research contribution.

To calculate inter-transaction distances, transaction attributes are extracted into feature vectors. Previous research has relied on SQL log-mining for transaction feature extraction, whereas this research proposes to use simple server-side extensions instead. Regular-expression based SQL log mining is error-prone, and parsing SQL text may require parser duplication. Using server-side extensions, no (re)parsing of a SQL statement is required beyond the one initiated upon a statement's submission. A MySQL implementation demonstrates that server-side feature extraction is feasible. Similar infrastructure already exists in most modern DBMS engines (Oracle, SQL Server, PostgreSQL, and so on), and hence the solution has wider applicability.

2 SQL TRANSACTION CLUSTERS

SQL transactions within a cluster are similar (but not identical), and transactions in different clusters are dissimilar. Cluster determination is a three-step process. First, certain distinguishing attributes (called *features*) are extracted from a transaction, and an n-element *feature vector (FV)* is formed. Second, the distance between two transactions—defined to be the distance between their feature vectors—is computed using a distance function. Third, a clustering algorithm uses the feature vectors and the distance function to determine clusters.

2.1 Feature vector construction

In this research, extracting the following transaction features proved adequate.

(1) Statement type: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `COMMIT`, `ROLLBACK`, `BEGIN`, and so on.
(2) Table name(s)—possibly empty—referenced in the statement in 'schema.table' format.
(3) Counts associated with table names indicating frequency.

For other applications, different or additional features may need to be extracted.

A transaction X's feature vector $FV(X)$ is a concatenation of four sub-vectors FV_S, FV_I, FV_U, and FV_D for the four major SQL statement types SELECT, INSERT, UPDATE, and DELETE, respectively. All of the four sub-vectors are computed similarly, and therefore, only the construction of FV_S is described.

FV_S is of length n—the total number of tables in the OLTP system's schema, where each table is schema-qualified, and occupies a specific position in the vector to enable cross-feature vector comparisons. If a table T is referenced by *all* of the SELECT statements in a transaction a total of k times ($k \geq 0$), then the vector element for T inside FV_S has value k.

FV_I, FV_U, and FV_D are computed similarly from all of the INSERT, UPDATE, and DELETE statements in a transaction.[1]

Listing 1: Transaction X_1

```
SELECT C_ID FROM Customer WHERE C_ID_STR = '50665'
SELECT * FROM Customer WHERE C_ID = 50665
SELECT * FROM Airport, Country WHERE AP_ID = 180 AND
    AP_CO_ID = CO_ID
SELECT * FROM Frequent_Flyer WHERE FF_C_ID = 50665
UPDATE Frequent_Flyer SET FF_IATTR00 = -14751,
    FF_IATTR01 = 8902 WHERE FF_C_ID = 50665 AND
    FF_AL_ID = 1075
UPDATE Customer SET C_IATTR00 = -14751, C_IATTR01 =
    89025 WHERE C_ID = 50665
COMMIT
```

Consider the transaction X_1—taken from the SEATS workload of [2]—shown in Listing 1.

- $FV_S(X_1) = [2, 1, 1, 1]$ because SELECT statements in X_1 refer to *Customer* table twice, and the other three tables once each.
- $FV_U(X_1) = [1, 1]$ because UPDATE statements in X_1 refer to two tables once each.
- $FV_I(X_1) = FV_D(X_1) = [\ ]$ because there are no UPDATE or DELETE statements in X_1.
- $FV(X_1) = [2, 1, 1, 1] + [\ ] + [1, 1] + [\ ] = [2, 1, 1, 1, 1, 1]$

Imagine a second transaction X_2 similar X_1 that references *Customer* only once. $FV(X_2)$ will be $[1, 1, 1, 1, 1, 1]$.

2.2 Angular cosine distance

Angular cosine distance, henceforth *ACD*, measures the distance between two transactions—in particular, between their feature vectors. For two n-dimensional vectors $\mathbf{A}$ and $\mathbf{B}$, each with indices $0, 1, \cdots n-1$, and with non-negative values, *ACD* is defined as follows [21].

$$ACD(\mathbf{A}, \mathbf{B}) = \frac{2}{\pi}\left(\cos^{-1}\left(\frac{\sum_{i=0}^{n-1} \mathbf{A}_i\mathbf{B}_i}{\sqrt{\sum_{i=0}^{n-1} \mathbf{A}_i^2}\sqrt{\sum_{i=0}^{n-1} \mathbf{B}_i^2}}\right)\right) \quad (1)$$

ACD is a distance measure or *metric*, and furthermore is unitary or normalized: $0.0 \leq ACD(\mathbf{A}, \mathbf{B}) \leq 1.0$. Because *ACD* distances are unitary, a closeness threshold *Eps* (for example, 0.2) can be defined so that two transactions at most *Eps* apart are considered 'close'; otherwise, they are declared 'far'. Normalized distance functions enable easy similarity definition: $similarity = 1.0 - distance$. 'Close' transactions have 0.8 (or 80%) similarity—something even a non-expert can understand.

For X_1 and X_2 of Section 2.1, $ACD(X_1, X2) = ACD([2, 1, 1, 1, 1.1], [1, 1, 1, 1, 1, 1]) = 0.197$, or they are $1.0 - 0.197 = 0.803$ (80.3%) similar which seems intuitively correct.

[1] INSERT, UPDATE, and DELETE statements can also have embedded SELECT queries, and those are handled similarly to the way FV_S is.

Let $FV(X_3) = [1, 1, 1, 1, 1, 0]$. X_3 does not update the *Customer* table, and hence the 0. X_3 is somewhat dissimilar to X_1 and X_2, and indeed, $ACD(X_1, X_3) = 0.295$, or they are only 70.5% similar which again seems intuitive. With the closeness threshold *Eps* set to 0.2, X_1 and X_2 will be considered close, and may end up in the same cluster, whereas X_1 and X_3 will not belong to the same cluster.

2.3 DBSCAN parameter tuning

An open-source DBSCAN implementation [16]—instrumented to use *ACD*—performs transaction clustering. (By default, it uses the Euclidean distance function.) The author did not consider other clustering algorithms because DBSCAN has been found adequate for transaction clustering previously [11, 23].

DBSCAN's two tunable parameters *Eps* and *minPts* define *density*. A hyper-sphere of radius *Eps* with at least *minPts* points inside is considered *dense*. *ACD* does not eliminate hand-tuning DBSCAN parameters, but provides twofold help.

- *Eps* = 0.2 means that *independent of workload*, two transactions have to be at least 80% similar before they can belong to the same cluster. With such unnormalized distance functions as the Euclidean, *Eps* values are workload dependent.
- For many workloads, *ACD*-based DBSCAN clustering is not very sensitive to the two parameter values, and a good starting point is (*Eps*, *minPts*) = (0.2, 10). (Section 5.2.)

3 MySQL EXTENSIONS FOR TRANSACTION FEATURE EXTRACTION

Feature extraction uses the data from three in-memory tables shown in Fig. 1 that contain transaction-level, statement-level, and table-level information. The three tables will henceforth be referred to by the acronyms *e_t_h*, *e_s_h*, and *e_s_t*.

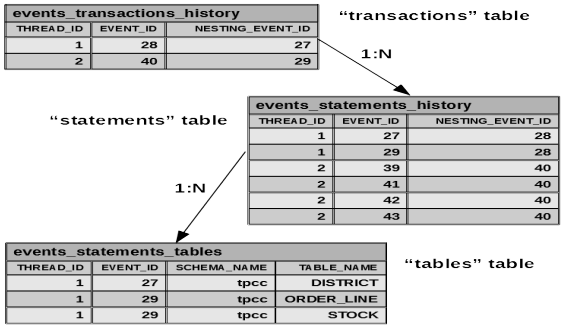


Figure 1: Relationships between the three tables.

Each completed transaction appears as a row in *e_t_h*, and the statements within are captured in *e_s_h* (1:N relationship). The newly added *e_s_t* table contains zero or more rows for each statement present in *e_s_h*—one for each distinct table reference in that statement (another 1:N relationship). Such statements as COMMIT and ROLLBACK do not refer to any tables, and therefore, have no presence in *e_s_t*. The columns in Fig. 1 only capture inter-table relationships. The other columns added to the three tables—not explicitly shown in Fig. 1—capture transaction-, statement-, and table-level statistics.

Using a SELECT query involving multi-way joins among *e_t_h*, *e_s_h*, and *e_s_t* tables, such information as transaction text; statement type; statement text; statement run-time; transaction

run-time; table names appearing in statement and their counts; number of rows examined; locking times; and so on is easily extracted.

Tables similar to the ones depicted in Fig. 1 already preexist (or can be easily created) in SQL Server [10], Oracle [12], and PostgreSQL [13], and therefore, server instrumentation of the kind described in this section is possible in those products.

4 SYSTEM ARCHITECTURE

A prototype SQL transaction clustering system has been implemented as depicted in Fig. 2. The Linux KVM virtual machine represents a hosted environment running a customer application (OLTP-Bench and Sysbench workloads during experimentation). The top MySQL instance within the Linux KVM has been instrumented as described in Section 3 to enable transaction-level data collection and feature extraction. The Windows 10 machine contains data processing components, including those performing transaction clustering and classification. (A second hosted application would require its own data processing node.)

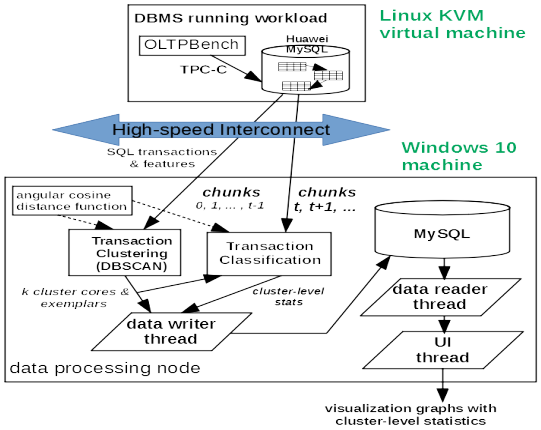


Figure 2: Architecture of a transaction clustering system.

A SQL query runs every 5 seconds on the Linux machine, and performs data collection. (The interval ensures minimal data collection overhead, but is a system parameter.) Each 5-second chunk includes feature and non-feature data: SQL statements; statement types; table names; table counts; statement durations; transaction duration; lock times; and rows examined by statements. An epoch-style Linux timestamp is associated with each chunk, and the resulting time-series is shipped to the data processing node where chunk-based iterators process it.

The first t chunks are used to perform transaction clustering. During experimentation, the first 30 seconds worth of transactions ($t = 6$) were found adequate to determine transaction clusters, but t is a system parameter. OLTP workloads do not have ad-hoc queries, and so clusters, once formed, should not change. If that assumption is violated, DBSeer's online implementation of DBSCAN can be used [5], but we leave that for future work.

Once clusters form, the rest of the streaming transactions are simply classified. The average distance (computed using *ACD*) from a transaction X to a cluster's exemplars is calculated, and X is assigned to the cluster for which that average distance is minimum, as long as that minimum value is no more than a threshold (say 0.2). If the threshold is exceeded, X is declared an outlier.

Simple roll-up operations compute cluster-level statistics from transaction statistics as long as transactions are not outliers. If necessary, data about outliers can be captured and processed similarly.

5 CLUSTERING EXPERIMENTS

These experiments demonstrate that *ACD*-based DBSCAN is effective at finding transaction clusters, and is not very sensitive to *Eps* and *minPts* parameter values. Workload consists of OLTP-Bench [2] and Sysbench [6]. Out of OLTP-Bench's 15 workloads, the author was able to run 11.[2] Clustering effectiveness requires expected cluster counts which OLTP-Bench already provides, and were manually determined for Sysbench. In Tables 1 and 2, expected cluster counts are indicated in brackets after the workload names. Actual cluster counts are not always integral because each value is an average of 5 runs, and DBSCAN sometimes produces slightly different cluster counts for different samples.

5.1 *ACD*-based DBSCAN

ACD-based DBSCAN with (*Eps*, *minPts*) = (0.2, 10)—henceforth, $ACD(0.2, 10)$—is an excellent starting point for clustering database transactions. Table 1 captures $ACD(0.2, 10)$'s performance against a baseline provided by the well-known Euclidean distance function. In the Euclidean baseline, *Eps* and *minPts* values are set using a heuristic provided by the DBSCAN authors in a subsequent paper [14]. We will term the resulting baseline *SEKX* after the author names. The heuristic itself works as follows. Let the *dimensionality* of a workload—defined to be maximum feature vector length—be *DIM*. Then:

- Heuristic value of $Eps = (2 * DIM) - 1$
- Heuristic value of $minPts = (2 * DIM)$

$ACD(0.2, 10)$ handily outperforms *SEKX* in 8 out of 13 workloads, and equally important, is never worse than *SEKX* in the remaining 5 workloads. For 9 workloads, *SEKX* puts all of the transactions into single clusters, and hence is ineffective.

The following observations—cross-referenced in the 'Comments' column of Table 1—provide reasons why $ACD(0.2, 10)$'s cluster counts are slightly off in a few cases.

(1) Epinions cluster count is off by 1 because a transaction type selects from *Review* and *Trust* tables separately, and another type selects from their joined version. Both end up in the same cluster because the feature vector construction in Section 2.1 currently does not distinguish them.
(2) YCSB cluster count is off by 1 because two of the transaction types have point and range selects, but are otherwise identical, and that difference is currently not captured as a feature.
(3) AuctionMark and SEATS produced the correct cluster counts with $ACD(0.15, 10)$ and $ACD(0.15, 15)$, respectively.

Observations 1 and 2 suggest possible features that can be extracted, and added to the feature vector.

5.2 Sensitivity analysis of *ACD*

When used along with *ACD*, DBSCAN is not very sensitive to various *Eps* and *minPts* values. Table 2 captures *ACD* results for 6 sets of parameters. Approximately, cluster membership criterion becomes more stringent as one reads across a row, and hence cluster counts can be expected to diminish from left to right. As can be seen, *ACD* is not very sensitive to parameter values for most of

[2]Wikipedia turns out to be hard to cluster because two of the transaction types have frequencies of only 0.07% each, and are rarely present in samples. One can ignore 'Wikipedia' results, but they are included for completeness.

Table 1: *ACD* (0.2, 10) versus *SEKX* ($2 * DIM - 1$, $2 * DIM$)

Workload & expected cluster count	DIM	Avg. cluster count		Comments
		ACD	*SEKX*	
AuctionMark (9)	9	9.4	1	Observation 3
Epinions (9)	2	8	1	Observation 1
SEATS (6)	8	7	1	Observation 3
SIBench (2)	1	2	2	
SmallBank (6)	5	6	1	
sysbench_ro (10)	1	10	10	
sysbench_rw (10)	5	10	10	
TATP (7)	3	7	1	
TPC-C (5)	10	5	1	
Twitter (5)	2	5	1	
Voter (1)	4	1	1	
Wikipedia (5)	7	2	1	Footnote 2
YCSB (6)	2	5	1	Observation 2

the workloads. Even when it is (AuctionMark and sysbench_rw; and to a lesser extent SEATS and TPC-C), graceful degradation is observed. Therefore, it is not crucial for a DBA to get the values of *Eps* and *minPts* spot on, and any reasonable values should perform respectably well. 'Sysbench_rw' is tricky to cluster for its highly symmetrical transactions—all of the transactions are roughly equidistant from all of the other transactions—but two *ACD* configurations perform well.

Table 2: *ACD* with different *Eps* and *minPts* values.

Workload & expected cluster count	Avg. cluster count					
	ACD (.20,10)	*ACD* (.15,10)	*ACD* (.15,15)	*ACD* (.10,10)	*ACD* (.10,15)	*ACD* (.05,20)
AuctionMark (9)	9.4	9.4	5.4	8.2	5.4	4
Epinions (9)	8	8	8	8	8	8
SEATS (6)	7	7.4	6	8.4	7	5.2
SIBench (2)	2	2	2	2	2	2
SmallBank (6)	6	6	6	6	6	6
sysbench_ro (10)	10	10	10	10	10	10
sysbench_rw (10)	10	9	0.6	0	0	0
TATP (7)	7	7	7	7	7	7
TPC-C (5)	5	6	5.8	6	5.8	4.8
Twitter (5)	5	5	5	5	5	5
Voter (1)	1	1	1	1	1	1
Wikipedia (5)	2	2	2	2	2	2
YCSB (6)	5	5	5	5	5	5

6 PERFORMANCE TROUBLESHOOTING USING TRANSACTION CLUSTERING

Experiments in this section demonstrate how cluster-level performance monitoring simplify debugging of several real-life problems faced by cloud DBA's.

6.1 Cluster signatures

OLTP-Bench's SmallBank workload simulates some operations of a bank, and produces the cluster signatures shown in Fig. 3 with 10-terminal workload, and scale-factor of 1. The three sub-plots capture the average values of the three transaction-cluster-level metrics: lock time (in ms); number of rows examined; and TPS

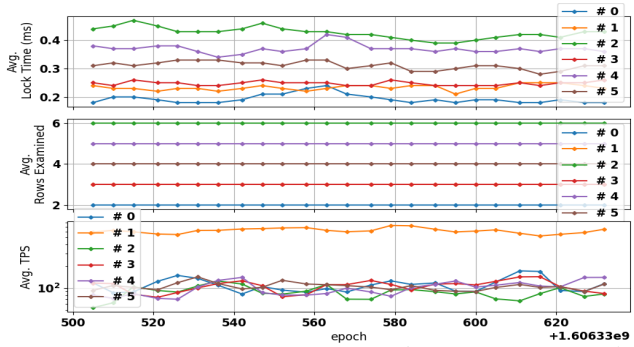


Figure 3: SmallBank's cluster signatures (10-terminal workload).

(transactions/sec). Each sub-plot contains six lines—one for each of the transaction clusters identified.[3]

From SmallBank cluster signatures, a cloud DBA can learn several things about the workload. First, six clusters means that there are six types of transactions in SmallBank—as OLTP-Bench confirms. Second, cluster signatures are discernible. Third, small signature variations exist because each data point aggregates 5 seconds worth of transactions (which themselves execute in a multi-tasking environment). Fourth, all of the transactions in a given cluster examine the same number of rows—a SmallBank peculiarity. Fifth, cluster exemplars reveal that the only cluster with read-only transactions is cluster 1—explaining its its highest TPS values. Sixth, when 100 terminals generate workload, the same six clusters form (graphs not included to save space), thereby confirming that clusters are load independent—a practically useful promise of clustering.

6.2 Identification of transaction rollbacks

Transaction rollbacks are normal DBMS occurrences, but sometimes their frequencies become problematic. If rollbacks are limited to a transaction type, cluster-level monitoring helps because unexpected additional clusters form.

The TPC-C benchmark [20] has five well-known transaction types. Rollbacks are demonstrated using the *Payment* transaction by modifying its code such that after submitting 2 out of its 7 statements, it rolls back with 20% probability—simulating a problematic high-frequency rollback. To make example even more realistic, a second rollback—simulating a normal and rare DBMS occurrence—happens after the sixth statement with 0.1% probability (overall probability $0.8 \times 0.1 = 0.08\%$). Because the problematic rollbacks are numerous, they should form their own cluster, whereas the normal rollbacks should not.

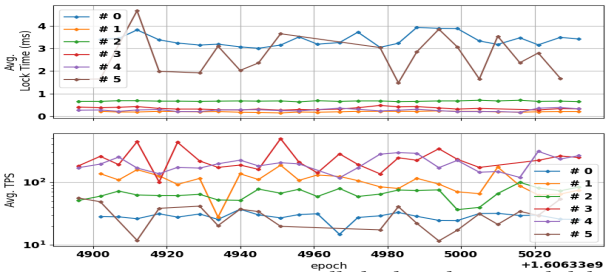


Figure 4: *Payment* transaction rolls back with 20% probability causing an unexpected sixth cluster (Cluster 1) to appear.

[3]The second subplot contains only five lines because clusters 1 and 3 examine 3 rows each, and the plotting software cannot distinguish two overlapping lines.

The modified TPC-C benchmark produces the results shown in Fig. 4. Usually, TPC-C produces five clusters, but six are found. A sample exemplar from Cluster 1 reveals the 'incomplete' *Payment* transaction with a telltale `rollback` issued after two statements.

```
UPDATE WAREHOUSE SET W_YTD = W_YTD + 1704.68994140625
    WHERE W_ID = 2
SELECT W_STREET_1, W_STREET_2, W_CITY, W_STATE, W_ZIP,
    W_NAME FROM WAREHOUSE WHERE W_ID = 2
rollback
```

The normal (rare) DBMS rollbacks (0.08% probability) do not form a cluster because they do not meet DBSCAN's density requirement mentioned in Section 2.3. High-frequency rollbacks are difficult to identify if cluster-level statistics are not kept. Applications often resubmit transactions in case of rollbacks, and users only notice and wonder about degraded performance. An incident report would cause DBA's or programmers to dig through voluminous logs to even begin suspecting a culprit.

6.3 Performance drift

Performance drift refers to a situation in which one (or just a few) cluster's performance drifts from its norm. Many situations can cause performance drifts. Here is a typical one: A DBA might forget to reinstate an index that (s)he deliberately dropped during a bulk load operation.

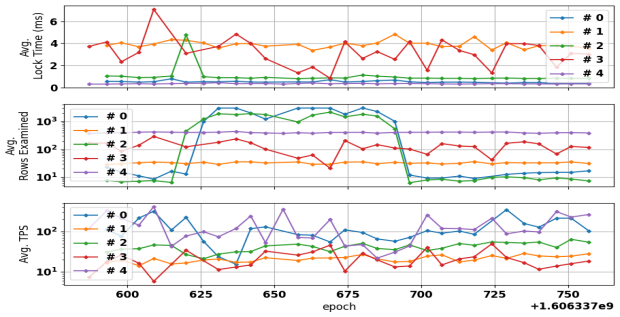


Figure 5: An index made invisible during [615, 685].

To simulate a performance drift, a secondary index (used by two out of the five TPC-C transactions) on the *CUSTOMER* table is made invisible[4] to the query optimizer during a portion of the run. When the index is made unavailable, the query optimizer has to use table scans instead of index seeks. As can be seen in Fig. 5, average row counts show dramatic increases (drifts) for clusters 0 and 2 during the interval [615, 685].[5]

When the performance of only one cluster drifts, objects related to only that cluster (e.g., tables, indexes, statistics) are good starting points for debugging. Without cluster-level statistics, such a diagnosis may require considerably more work.

6.4 System-wide performance problem

System-wide performance problems are caused by such things as a failed network card, operating system reboot, failed disk, and runaway process hogging CPU's. If all of the clusters experience simultaneous degraded performances, a system-wide issue may be the cause. One such situation is created using a CPU-hogging program that spawns as many processes as the number of CPU cores on the computer (8), and then making them run infinite 'while' loops—thereby creating a CPU bottleneck.

[4]MySQL command used was: `ALTER TABLE CUSTOMER ALTER INDEX IDX_CUSTOMER_NAME INVISIBLE;`

[5]As an aside, if an invisible index makes no difference to any cluster's performance, it might be safe to drop—the reason why that feature was added to MySQL.

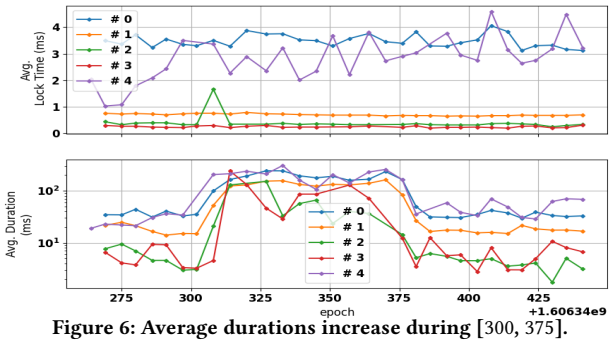


Figure 6: Average durations increase during [300, 375].

In the resulting graphs shown in Fig. 6, during the interval [300, 375], average durations of all of the clusters show unmistakable jumps. After about 375, when the offending program is killed, all five average durations return to their baseline values. Interestingly, average lock times are largely unaffected, indicating that for the few transactions that did manage to execute during CPU saturation, lock time did not take a hit. Such observations should provide the DBA a good starting point to formulate a hypothesis before beginning a detailed investigation.

6.5 Bottleneck analysis

Cloud applications run on pre-provisioned VM's. Because application behaviour is relatively unknown, a bottleneck may develop—in CPU, memory, disk I/O, network I/O, and so on. Furthermore, bottlenecks may vary by transaction types. Non-cloud DBA's are used to monitoring such operating system-level performance counters as *vmstat*, *iostat*, and *netstat* in Linux for bottleneck identification, but cluster-level statistics offer a complementary method that can provide additional help.

To study whether a VM has sufficient memory, its memory is reduced on the fly from 16 GB to 3 GB while TPC-C workload runs. Such a drastic change in memory allocation is only for demonstration: typical changes should be much smaller.

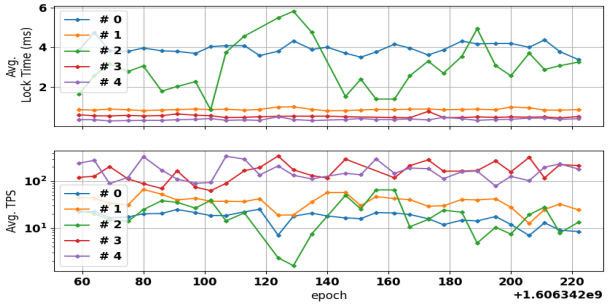


Figure 7: Memory reduces from 16 GB to 3 GB at timestamp 120.

In the results captured in Fig. 7, memory reduction happens at timestamp 120 onward. The average TPS values before and after that interval show no discernible changes. There is a noticeable drop at 120 as the operating system seems to adjust to the new memory setting, but soon, normal service resumes. The 'Avg. lock time' metric is also mostly unaffected, and therefore, one can conclude that this VM is well-provisioned for memory.

In the next variation, CPU is constrained. Changing CPU count in KVM requires a machine restart, and therefore, an approach similar to the one in Section 6.4 is taken, except that 7 out of the

8 cores are kept busy running infinite 'while' loops. The results appear in Fig. 8.

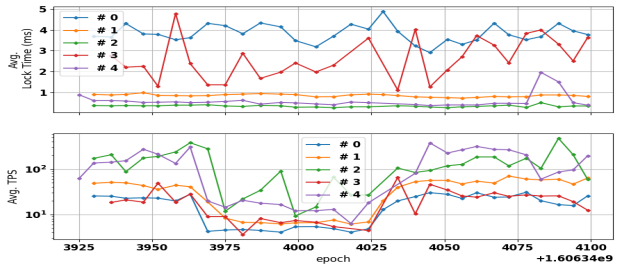


Figure 8: CPU saturation during the interval [3960, 4025].

The CPU bottleneck spans the interval [3960, 4025] during which reduced 'Avg. TPS' values are visible. The highest 'Avg. TPS' values are for clusters 2 and 4 (read-only transactions *Stock-Level* and *Order-Status*, respectively). Outside of the CPU bottleneck, those values are somewhat close, but during the bottleneck, *Order-Status* transaction's performance takes a bigger hit (Y-axis is log-scale) suggesting *Order-Status* is much more sensitive to CPU than *Stock-level* is in this environment. If *Order-Status* is deemed important (say because customers check statuses their orders often), it may make sense to over-provision for CPU rather than for memory if a choice is to be made between the two.

7 RELATED WORK

Identifying transaction clusters from SQL text arriving at a database server is important for two reasons. First, applications deployed in cloud environments are often web-applications [17] using object-relational mappings to submit SQL queries, and do not use stored procedures. Second, as noted by Stonebraker *et al.* [19], because of SQL's 'one language fits all' approach, transaction code may use a mix of stored procedures, prepared statements, and Java/C++/C# code.

Clustering itself is a broad and well-studied topic [22]. SQL query clustering and classification has been studied under two granularities: query-level and transaction-level. SQL query features previously tried include terms in `SELECT`, `JOIN`, `FROM`, `GROUP BY`, and `ORDER BY` clauses, table names, column names, normalized estimated execution costs [7, 9]; and features have been converted into vectors, graphs, or sets [7, 18]. As a general observation, fewer features suffice in self-similar OLTP workloads; ad-hoc workloads require more features. Such distance functions as cosine, Jaccard, and Hamming have been tried for clustering SQL queries [1, 9, 18], although only the Euclidean has been tried at the transaction level before [5]. Before this research, feature extraction has mined SQL text from DBMS logs [9, 18], or MaxScale proxy server [11]; server-side feature extraction is novel.

8 CONCLUSIONS AND FUTURE WORK

This research makes a case that in addition to user, table, index, and schema level monitoring provided, DBMS's should start to provide transaction-cluster-level monitoring. In applications deployed in the cloud, and for OLTP workloads, that additional level simplifies debugging of performance problems: unexpected transaction rollbacks, performance drifts, bottleneck identifications, and so on. Angular cosine distance-based DBSCAN is an improvement over Euclidean-based DBSCAN with *SEKX* heuristic [14] (better clusters and simplified DBSCAN parameter tuning).

Future work may investigate the following features for transaction clustering: column names to possibly identify index issues; predicate types (point queries vs. range queries) and counts; access paths used; isolation level; number of sorts; join tables; and so on. Multiple cluster-level signatures may help because an application may have distinct 'peak' and 'off-peak' behaviours. Whether *ACD* is suitable with such clustering algorithms as BIRCH [24] and k-means [8] remains to be seen. Server-side feature extraction can be attempted in other modern database systems using minor extensions to preexisting scaffoldings.

ACKNOWLEDGMENTS

Matthew Van Dijk implemented the server-side extensions needed for transaction feature extraction. Anonymous reviewers provided valuable feedback.

REFERENCES

[1] Rakesh Agrawal, Ralf Rantzau, and Evimaria Terzi. 2006. Context-sensitive ranking. In *Proceedings of the SIGMOD 2006 Conference*. 383–394.
[2] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.
[3] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*. 226–231.
[4] Junhao Gan and Yufei Tao. 2015. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *Proceedings of the SIGMOD 2015 Conference*. 519–530.
[5] GitHub. 2020. *DBSeer*. Retrieved September 17, 2020 from https://github.com/barzan/dbseer
[6] GitHub. 2020. *sysbench*. Retrieved August 5, 2020 from https://github.com/akopytov/sysbench
[7] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130.
[8] Stuart P. Lloyd. 1982. Least squares quantization in PCM. *IEEE Trans. Inf. Theory* 28, 2 (1982), 129–136.
[9] Vitor Hirota Makiyama, Jordan Raddick, and Rafael D. C. Santos. 2015. Text Mining Applied to SQL Queries: A Case Study for the SDSS SkyServer. In *SIMBig*.
[10] Microsoft. 2019. *System Dynamic Managemenet Views*. Retrieved August 4, 2020 from https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/system-dynamic-management-views?view=sql-server-ver15
[11] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proceedings of the SIGMOD 2013 Conference*. 301–312.
[12] Oracle. 2020. *About Dynamic Performance Views*. Retrieved September 22, 2020 from https://docs.oracle.com/cd/B19306_01/server.102/b14237/dynviews_1001.htm#i1398692
[13] PostgreSQL. 2020. *The Statistics Collector*. Retrieved August 4, 2020 from https://www.postgresql.org/docs/9.6/monitoring-stats.html
[14] Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 1998. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. *Data Min. Knowl. Discov.* 2, 2 (1998), 169–194.
[15] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.* 42, 3 (2017), 19:1–19:21.
[16] Scikit Learn. 2019. *DBSCAN*. Retrieved August 2, 2020 from https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html
[17] Alexandre Verbitski *et al.* 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the SIGMOD 2017 Conference*. ACM, 1041–1052.
[18] Gökhan Kul *et al.* 2018. Similarity Metrics for SQL Query Clustering. *IEEE Trans. Knowl. Data Eng.* 30, 12 (2018), 2408–2420.
[19] Michael Stonebraker *et al.* 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the VLDB 2007 Conference*. 1150–1160.
[20] Transaction Processing Performance Council 1992. *TPC-C*. Retrieved September 22, 2020 from http://www.tpc.org/tpcc/
[21] Wikipedia. 2019. *Cosine similarity*. Retrieved August 11, 2020 from https://en.wikipedia.org/wiki/Cosine_similarity
[22] Dongkuan Xu and Yingjie Tian. 2015. A Comprehensive Survey of Clustering Algorithms. *Annals of Data Science* 2 (2015), 165–193.
[23] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. DBSherlock: A Performance Diagnostic Tool for Transactional Databases. In *Proceedings of the SIGMOD 2016 Conference*. 1599–1614.
[24] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1997. BIRCH: A New Data Clustering Algorithm and Its Applications. *Data Min. Knowl. Discov.* 1, 2 (1997), 141–182.

Short Paper

Revisiting Multidimensional Adaptive Indexing*

Anders Hammershøj Jensen
Aarhus University

Frederik Aarup Lauridsen
Aarhus University

Fatemeh Zardbani
Aarhus University

Stratos Idreos
Harvard University

Panagiotis Karras
Aarhus University

ABSTRACT

Modern applications require managing large data in main memory. *Adaptive indexing* allows for building an index incrementally in response to queries, rather than upfront; in its default form, it treats each attribute independently. However, several data exploration tasks involve multidimensional range queries. Two recent proposals, CKD and QUASII, address the need for multidimensional (especially spatial) adaptive indexing. Both adaptively build an augmented KD-Tree. Still, no previous work has compared these two methods to each other. We conduct the first experimental comparison of CKD and QUASII. Further, we propose a lightweight variant of CKD, the Lazy CKD, which performs data-driven along with query-driven actions for the sake of robustness, and a set of hybrid strategies that combine good convergence and low initialization cost. Our study on synthetic and real data and workloads shows that the enhanced variants of CKD have an advantage in terms of speed of convergence, yet QUASII may eventually achieve lower response times.

1 INTRODUCTION

Scientists and analysts need to query and explore large amounts of data in dynamic environments where new data arrive continuously, without building a full index in advance. This need calls for self-organizing DBMSs that eschew human administration. *Adaptive indexing* [8], such as *database cracking* [4, 6], accommodates this need by building and refining a main-memory index incrementally, in response to queries and arriving data. Cracking is applied within the select operator in a column-oriented database [7, 10]. A *stochastic* alternative [4] creates *random cracks* so as to perform robustly on skewed workloads.

Despite the intense interest in adaptive indexing, such methods for multidimensional data have been examined scantly. Recently, two solutions appeared: the QUery-Aware Spatial Incremental Index [12] (QUASII) and the Cracking KD-Tree (CKD) [5] (CKD). Both incrementally build an augmented KD-tree [1]; both conclude that their query response times converges to that of static approaches after processing a sufficient amount of queries; however, no comparison among these two works has been attempted.

In this paper, we conduct the first comparison of these two approaches [5, 12]; we also propose a lightweight CKD variant, the Lazy CKD (LCKD), incorporate stochastic cracking [4] strategies to improve robustness, and propose hybrid strategies that combine desirable traits of different solutions. We evaluate all methods on both synthetic and real datasets and workloads.

*The two first authors contributed equally to this work.

2 1D ADAPTIVE INDEXING

Cracking [6] progressively partitions and sorts a column by quicksort while answering range queries. A partition containing one or both query range bounds is further split, or *cracked*, up to a minimum size. A partition containing the entire interval is split into three. Cracking directly into three partitions is possible [6], yet cracking into two partitions twice instead yields better results [13]. Listing 1 presents the basic cracking algorithm.

Listing 1: Crack in two [6].

```
1  def crack_in_two(pivot p, value low, value high):
2    x1 ← point at position low
3    x2 ← point at position high
4    while (position (x1) < position((x2)):
5      if (value(x1) < p):
6        x1 ← point at next position
7      else:
8        while (value(x2) ≥ p &&
9               position (x2) > position(x1)):
10              x2 ← point at previous position
11       exchange(x1, x2)
12       x1 ← point at next position
13       x2 ← point at previous position
```

Stochastic Cracking [4] adds *data driven* cracking actions to standard *query driven* ones, which may perform poorly on skewed workloads. The Data Driven Center (DDC) algorithm cracks the partition where a query bound lies at the median recursively, until obtaining a sufficiently small partition, whereupon it cracks at the query bound. Data Driven Random (DDR) avoids median-finding by choosing a random pivot. The DD1C and DD1R variants perform only one median or random crack. Still, these strategies retain the overhead of query-driven cracking. To ameliorate it, Materialization-based DD1R (MDD1R) [4] cracks the piece in which a query bound lies only on a single random pivot and materializes the result. Progressive MDD1R [4] shares the burden across queries, allowing crackin to be partially completed. Refined variants use the median of a sample set instead of a single random pivot, with performance gains [18].

Adaptive Merging [2] splits the data into arbitrary initial partitions, from which it progressively extracts query results into to a final sorted partition by incremental mergesort; it achieves faster convergence than cracking at the cost of high initialization cost [8]. A hybrid combination of the two [8] applies cracking on initial partitions and sorting on the final one to achieve both lightweight initialization and quick convergence.

Multidimensional Cases. A first study in adaptive multidimensional index structures [16] was about reorganizing, rather *building*, data-oriented hierarchical indexes, in response to a workload, so as to improve performance. Recently, QUASII [12] and CKD [5] extended adaptive indexing to the multidimensional case, by applying cracking on one dimension per tree level to construct a KD-tree-like structure. To our knowledge, these works have not been compared. Next, we discuss them in detail.

3 MULTI-D ADAPTIVE INDEXING

Augmented KD-tree. Both QUASII [12] and CKD [5] build their indexes on top of an augmented KD-tree [1], progressively redistributing data from the root node to newly created children

nodes while processing queries, allowing for a *variable number of children per node*. As new children nodes are added irregularly, the tree may lose the property of being *balanced*.

Listing 2: QUASII query [12].

```
1  def query(query q, data D, slices S, result R):
2    S' ← ∅          // to store newly created (refined) slices
3    dim ← S[0].l    // current level/dimension of slices in S
4    i ← binarySearch(S, lower(q[dim]))
5    while (i < |S| and lower(S[i].box[dim]) ≤ upper(q[dim])):
6      if q ∩ S[i].box = ∅ then continue
7      S'' = refine(S[i], q, D)
8      for each s ∈ S'':
9        if q ∩ s.box ≠ ∅:
10         if s.l is the bottom level:
11           for j ∈ s.ids:
12             if D[j] ∩ q ≠ ∅ :
13               R ← R ∪ D[j]
14         else:
15           if |s.children| == 0 :
16             createDefaultChild (s)
17           query(q, D, s.children, R)
18     S' ← S' ∪ S''
19     i ← i + 1
20   S ← S ∪ S'
21   sort (S)
```

QUASII [12] comprises a hierarchical index structure of depth equal to the dimensionality d; the index starts off as a root node containing all data objects, unsorted, and grows while processing range select queries. A bottom-level node may be split if it contains more than τ objects; thus, QUASII slices a space with n objects up to $r = \left\lceil \sqrt[d]{n/\tau} \right\rceil$ times in each dimension; at level ℓ above the bottom the threshold becomes $\tau[\ell] = r^{\ell}\tau$. Listing 2 shows how QUASII processes a multidimensional range query q; it finds the first slice hitting q by binary search (Line 4) along dimension (tree level) dim, on which slices are sorted along. It then scans and refines (i.e., slices further) all slices intersecting q (Lines 5–7), cracking them up to size τ, by both query bounds and artificial cracks. If a resulting slice at the bottom level intersects q (Line 9), qualifying data objects therein are appended to the result R (Lines 10–13); in case the slice is at an internal level, QUASII recursively queries that slice's children (Lines 15–17). After refining all relevant slices in level (dimension) dim, QUASII resorts the slices therein (Line 21) with respect to the lower bound of their bounding boxes. After recursively traversing, cracking, and resorting slices as needed, it returns the range query result R.

Listing 3: Refine [12].

```
1  def refine ( slice s, query q, data D):
2    if (|s| ≤ τ[s.l]):
3      return {s}
4    S ← ∅
5    t ← determineSliceType(s,q)
6    switch(t):
7      case both: S' ← sliceThreeWay(s, q, D)
8      case one: S' ← sliceTwoWay(s, q, D)
9      default: S' ← sliceArtificial(s, q, D)
10   for each s ∈ S':
11     if (|s| > τ[s.l] and q[s.l] ∩ s.box[s.l] ≠ ∅):
12       S'' ← sliceArtificial(s, q, D)
13       S ← S ∪ S''
14     else:
15       S ← S ∪ s
16   return S
```

Listing 3 illustrates the process that refines each slice s that intersects the query q along the examined dimension $s.l$ and exceeds the size threshold $\tau[s.l]$. QUASII cracks on any bound of q along dimension $s.l$ that lies within s (Lines 7–8); otherwise, if q contains s along $s.l$ (i.e., both bounds of q lie outside s), it slices based on an artificially introduced coordinate $c = \lfloor (x_l + x_u)/2 \rfloor$ (Line 9), where x_l (x_u) is the lower (upper) bound of s along $s.l$; it recursively slices further each produced slice that exceeds the τ size threshold and overlaps with q (Lines 11–12), otherwise adds it to the output (Line 15). Listing 4 presents the recursive procedure for such *artificial slicing*, which configured to handle multiple points having the same coordinate when cracking (Line 5).

Listing 4: Artificial slicing.

```
1  def sliceArtificial ( slice s, query q, data D):
2    if |s| ≤ τ[s.l]:
3      return [s]
4    c = ⌊(x_l + x_u)/2⌋
5    slices = crack(s, c, s.l)
6    ret = []
7    for s' ∈ slices:
8      ret = ret ∪ sliceArtificial(s', q, D)
9    return ret
```

QUASII represents each spatial data object using its lower coordinate only along any dimension. A slice is first defined by its cracking coordinates, or *cuts*, yet obtains its own *minimum bounding box* (MBB) embracing the spatial extent of each object therein, with overlapping among slice MBBs allowed. To avoid the MBB computation overhead, QUASII computes MBB bounds for a slice s only on the dimensions s has been fully refined along. During slice refinement and binary search, to capture any result whose representative corner point lies in an unrefined slice outside the query range, QUASII extends the lower coordinate of q by the maximum object extent in each unrefined dimension, as in [15]. QUASII was designed as an adaptive spatial index for data in 2 or 3 dimensions; in our experiments we test its performance on higher dimensionality too.

Cracking KD-Tree (CKD) [5] also lets an augmented KD-Tree grow through queries, and uses a minimum node size threshold τ. However, it assigns dimensions to tree levels in round-robin fashion by a *modulo* operation, allowing for multiple levels cracking on the same dimension. Listing 5 shows how CKD processes a multidimensional range query q on a slice s; if s is fully contained within q (Line 2), it extracts the contents of the given slice s, otherwise traverses any children of s (Lines 4–5); if s has no children and contains fewer than τ elements, a check of those vs. q yields the result (Lines 6–7). If none of the above is the case, CKD cracks the slice; it further queries resulting slices lying outside the query bounds on dim, to refine the index on other dimensions (Line 13), and those within the query bounds, to obtain results (Line 16).

Listing 5: CKD query.

```
1  def query(query q, slice s, dimension dim):
2    if ( isIncluded (s, q)):
3      return extractPoints (s, q)
4    if (s.children > 0):
5      return traverseTree (s, q, dim)
6    if (|s| ≤ τ):
7      return extractPoints (s, q)
8    slices ← crack(s, q, dim)
9    nextDim ← dim + 1 mod q.maxDim
10   for s' in slices:
11     s . add_slice (s')
12     if (s' ∩ q == ∅):
13       query(q, s', nextDim)
14     else
15       relevantSlice ← s'
16   return query(q, relevantSlice, nextDim)
```

Listing 6 shows the tree taversal procedure. CKD finds the first slice within the bounds of q by binary search (Line 3) and examines all slices within those bounds (Line 4–8), adjusting query bounds for the current dimension to fit the data in the slice (Line 6) and builds the result by querying those slices (Line 7).

Listing 6: CKD and LCKD traverse tree.

```
1  def traverseTree ( slice s, query q, dimension dim):
2    res ← []
3    i ← binarySearch(S, lower(q[dim]))
4    while (i < |s| and lower(s[i]) ≤ upper(q[dim])):
5      s' ← s[i]
6      q' ← adjustQueryToSliceBoundaries(q, s', dim)
7      res ← res ∪ query(q', s', nextDim)
8      i ← i + 1
9    return res
```

4 DISCUSSION AND ENHANCEMENTS

Lazy Cracking KD-Tree. The CKD [5] gratuitously refines all resulting slices after a crack on the query range according to query bounds. This gratuitous refinement is redundant; as it suffices to crack the pieces overlapping the query. We propose a variant that does so, the *Lazy Cracking KD-Tree* (LCKD). Listing 7 displays how LCKD processes a query, the change seen in Lines 10–13. In each dimension, LCKD only cracks partitions between query bounds, rather than cracking all pieces in excess.

Listing 7: LCKD query.

```
1  def query(query q, slice s, dimension dim):
2    if (isIncluded(s, q)):
3      return extractPoints(s, q)
4    if (s.children > 0):
5      return traverseTree(s, q, dim)
6    if (|s| ≤ τ):
7      return extractPoints(s, q)
8    slices ← crack(s, q, dim)
9    nextDim ← dim + 1 mod q.maxDim
10   for s′ in slices:
11     s.add_slice(s′)
12     if (s′ ∩ q != ∅):
13       relevantSlice ← s′
14   return query(q, relevantSlice, nextDim)
```

Intuitive Comparison We now now have three strategies to compare: QUASII, CKD, and LCKD. All start with a tree consisting of a single node, and progressively build a hierarchical index while answering queries. In QUASII, the tree height is equal to the number of dimensions. In CKD and LCKD, the maximum tree height is determined by the size threshold τ for cracking a partition. Figure 1 sketches the tree structures resulting after processing a single query under these strategies in the two dimensions; each tree layer corresponds to a different dimension. In CKD, there is an initial three-way crack along dimension x, followed by cracking all resulting pieces along query bounds on y. LCKD only cracks the x-partition relevant to the query along the bounds on y. The tree LCKD builds is less balanced than that of CKD, as successive queries may refine the index in the same region. Lastly, QUASII fully refines the entire part of the index relevant to the query in each dimension, making the tree wider, down to the minimal partition size τ [12]; CKD cracks the chunk relevant to the query, as well as pieces irrelevant to the query, on all dimensions according to query bounds, yet not necessarily down to final partitions; LCKD refines only the relevant part of the data along the query bounds in each dimension.

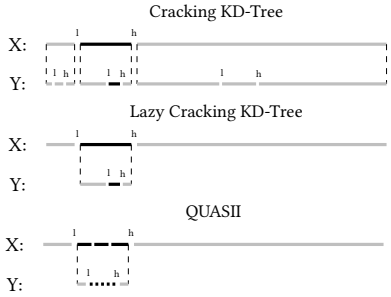


Figure 1: Three cracking strategies in action.

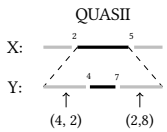


Figure 2: QUASII rigidity; Q1[(2,4)–(5,7)], Q2[(3,6)–(6,9)]

Figure 2 illustrates the problem that would arise in case QUASII did not perform full refinement. Assume a query on the range defined by lower left coordinates $(2, 4)$ and upper right coordinate $(5, 7)$, triggering a cracking of the x dimension on range $[2, 5]$ and the y dimension on range $[4, 7]$. Consider two points, $(4, 2)$ and $(2, 8)$, which belong in the x-range $[2, 5]$, but get separated by the y-interval $[4, 7]$. If a subsequent query requested the range from $(3, 6)$ to $(6, 9)$, we should crack the x dimension at 3, hence swap points $(2, 8)$ and $(4, 2)$, destroying the sorted order on y. To prevent such an eventuality, QUASII cracks exhaustively upon the first query on any data range. Given this rigidity of QUASII, we discuss *stochastic cracking* on LCKD only.

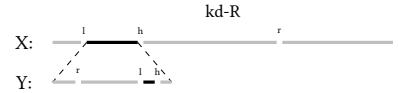


Figure 3: kd-R in action.

Multidimensional stochastic cracking As described in Section 2, we may spur query-driven cracking via data-driven cracks, to achieve faster convergence with a slight initialization overhead [4]. In [4], the DD1R strategy emerged as most commendable. We mend LCKD using DD1R in the multidimensional case; we call the resulting method kd-R, where R stands for *random*. In each dimension, when cracking a data segment, kd-R cracks at a random point in addition to the query bounds, as Figure 3 shows. Next, we examine other ways to improve upon LCKD.

Hybrid cracking We propose kd-HR, a *hybrid* strategy that aims to combine the fast convergence of QUASII and the low initialization cost of LCKD; kd-HR initially behaves as kd-R, yet switches to QUASII and stops creating new levels once the tree reaches a specified threshold. We design two kd-HR variants, depending on the nature of the threshold: kd-HR$_s$, with a threshold on node size, and kd-HR$_\ell$, with a threshold on tree height. Figure 4 shows the operation of kd-HR on the last level before switching to QUASII. kd-HR postpones QUASII-like operation until the data becomes sufficiently small (kd-HR$_s$) or the area in question has been refined enough times (kd-HR$_\ell$); this precaution should bring about faster convergence, as the tree stops growing further on branches switching to QUASII.

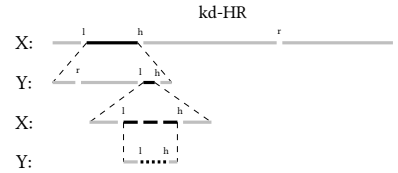


Figure 4: kd-HR in action.

5 EXPERIMENTAL STUDY

Here, we present our experimental study featuring QUASII, CKD, and their stochastic and hybrid variants. We implemented all methods[1] in C++ and compiled in g++ 7.4.0, and conducted experiments on a 10-core Intel Xeon CPU E5-2687W v3 machine at 3.10GHz with 396G RAM running Ubuntu 18.04.3 LTS.

ID	Name	Size	Distribution
0	Random	240000	Uniform distribution
1	Skyserver full	722711000	Skyserver data
2	Neuroscience	1000000	Neuronal data

Table 1: Datasets

Data. Table 1 lists our data sets. Random is a synthetic containing points distributed uniformly at random between 0 and 1 in each dimension; the default dimensionality is 2. The Skyserver data are downloaded from [14], a public astronomical data repository, by the CasJobs functionality. We chose two attributes, *declination*, dec, and *right ascension*, ra. To experiment with data

[1]The code is available at https://github.com/MULTIDAI/MultiDAI

of varying size, we construct trunacted versions of this data, selecting every ith point, $i \in \{2, 4, 8, \ldots, 512\}$. The Neuroscience dataset consists of 3-dimensional model of a neocortical column in a brain tissue with MBBs matched to neuronal axons.

Name	Size	Distribution
Random	10000	Random distribution
Sequential	1000	Queries along diagonal
Skyserver chronological	1000000	Skyserver workload
Random clusters	50000	Synthetic normal clusters

Table 2: Workloads

Workloads. Table 2 lists our query workloads. The first synthetic workload, Random, issues range queries at locations selected uniformly at random, with extent 1% of the value domain per dimension. The second synthetic workload, Sequential, issues a non-overlapping sequence of consecutive range queries along the diagonal of the domain of celestial coordinates, again with extent 1% of the value domain per dimension; as this workload explores the data space incrementally in small steps, the index constructed under its guidance never gets an opportunity to exploit previous indexing. Skyserver workloads derive from the SqlLog table [14], containing queries executed by scientists in nonrandom patterns, focusing on one sky area at a time [4]. We filtered the range selection predicates on declination dec and right ascension ra. We use three versions of this workload: chronological preserves the original order of queries; sorted on x contains the same queries, sorted on lower dec coordinate; sorted on size sorts them on total area of query range. The latter two workloads present a skewed pattern resembling the skewed sequential workload we apply on synthetic data. The Random cluster workload is synthetically generated for use with the Neuroscience data; queries belong to Gaussian clusters surrounding 5 randomly chosen centers in the range of the data values, with a maximum query volume of 0.01% of the whole data volume.

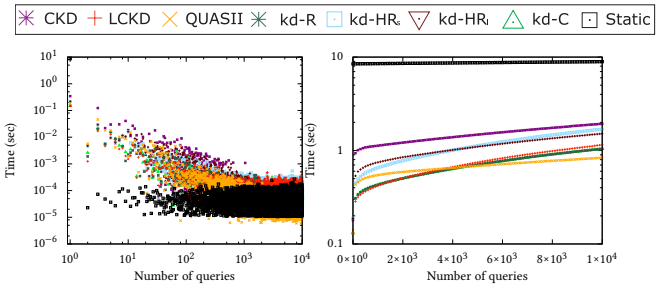
Compared methods. We juxtapose QUASII [12]; CKD [5]; LCKD; kd-R; kd-HR$_s$ switching to QUASII when nodes that are smaller than $s = 1000$; kd-HR$_\ell$ switching to QUASII after $\ell = 6$ levels; kd-C, a variation of kd-R that performs data-driven cracks on the center of the value domain; Static, a static implementation of a KD-tree. Previous works have already compared QUASII [12] to a static R-tree [3] and CKD [5] to a static KD-tree [1].

Tree	Synthetic	Skyserver	Neuroscience
QUASII	400	40	200
CKD	1000	170	400
LCKD	400	90	200
kd-R	400	80	200
kd-C	400	70	200
kd-HR	400	80	200

Table 3: Chosen τ values for synthetic and real data.

Parameter tuning. All methods require a τ parameter — the minimal cardinality of a data slice that may be further cracked. We conducted a series of experiments on the Random data (Table 1), with the Random and Synthetic workloads (Table 2). For each method, we chose as default the value of τ for which we observed best performance. Table 3 presents those choices. In size-based hybrid, kd-HR$_s$, we chose s through experimentation with several values; $s = 1000$ yielded the best performance.

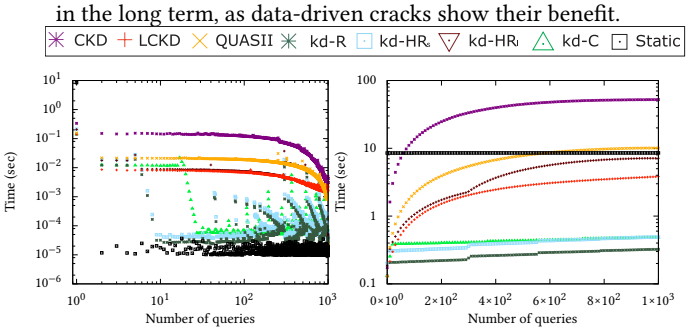
Random workload. We first examine the Random data with the Random workload. Figure 5a presents time per query. Evaluating the first query with Static counts for full index building. CKD is the slowest among adaptive methods in the first query, as it scans all data twice: once to crack on the first dimension, and again to crack the three resulting slices on the second dimension. QUASII comes second, while LCKD is the fastest. The initialization costs of stochastic and hybrid structures are almost identical, and slightly lower than that of QUASII. After the first query, CKD variants reduce time per query, yet LCKD converges faster than CKD. The time of QUASII falls intermediately, yet keeps falling further than LCKD. QUASII performs more work in early stages, yet achieves fast response times later on, traversing a shallower tree. As the workload evolves, fewer queries require additional indexing actions in QUASII, letting time per query fall to even less than that of Static. Figure 5b presents cumulative running time; the divergence between QUASII and the others is conspicuous; Static is much slower than the adaptive approaches; CKD incurs higher cumulative runtime than other adaptive structures.



(a) Time per query **(b) Cumulative time**

Figure 5: Results on Random workload.

A main takeaway from this experiment is that, over time, QUASII starts reaping the fruits of the index is has built more than other methods. We found this trend to be the same regardless of dataset: LCKD outperforms QUASII in the first queries at the expense of the quality of the index it builds, yet later queries do not benefit as much from the work done previously as they do with QUASII, and QUASII eventually outperforms LCKD. Unfortunately, the kd-HR hybrids present response times worse than LCKD, kd-R, and kd-C. We infer that the combination of kd-R and QUASII leads to a poor structure. Stochastic variants, kd-R and kd-C, present similar runtimes to LCKD, with an advantage in the long term, as data-driven cracks show their benefit.



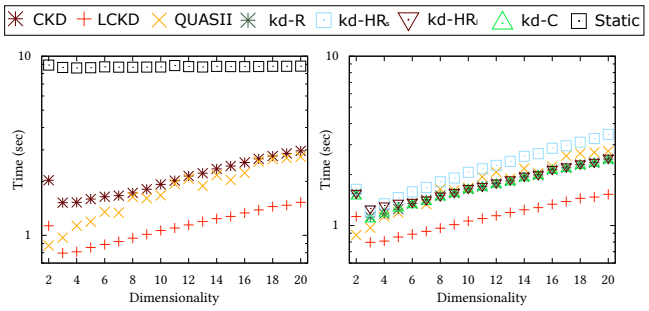
(a) Time per query **(b) Cumulative time**

Figure 6: Results on sequential workload.

Sequential workload. Now we turn to the Random data with the Sequential workload. Figure 6 shows our results. Unsurprisingly, CKD performs poorly. More surprisingly, in contrast to the preceding experiment, QUASII also performs poorly compared to LCKD; the thorough index refinement QUASII performs is a liability with the sequential workload. LCKD benefits from its lazy nature. This experiment reveals that the order of query execution has a significant effect on running time, due to the query-driven nature of these data structures: on a sequential workload, each new query processes an unindexed data region. Notably, kd-R performs best; its data-driven operation confers an advantage on this workload. The spikes in the plot arise when entering a region refined by data-driven cracks to a lesser extent. Interestingly, kd-C does not match the performance of kd-R: its center-based cracks prove to be insufficiently robust, vindicating

our choice to build hybrids on top of kd-R rather than kd-C. Yet those hybrids do not match the performance of kd-R either; their QUASII component appears to be a liability; this effect is apparent on kd-HR$_s$ and even more consequential in kd-HR$_\ell$.

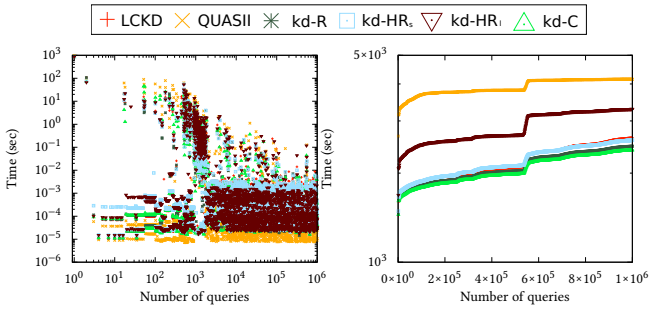
Effect of dimensionality. We now evaluate the impact of data dimensionality on the Random dataset and the Random workload. Figure 7 depicts our results in two plots for the sake of readability. As dimensionality grows, the cumulative time of QUASII rises drastically, due to the thorough refinement performed on *each* dimension. CKD and LCKD are less affected by dimensionality growth. LCKD presents a modest runtime growth with dimensionality. Surprisingly, the cumulative runtime of the KD-tree variants initially drops as dimensionality increases, as they gain from the indexing they perform. In the global trend, CKD incurs a heavier cumulative runtime burden, as additional dimensions beget a higher overhead than the benefit of cracking. QUASII, which performs well on random workloads on data of dimensionality 2, forfeits this advantage in higher dimensions. QUASII and QUASII-based hybrids, especially kd-HR$_s$, are affected by data dimensionality to a greater extent than LCKD, kd-R, and kd-C. We deduce that the QUASII strategy is detrimental on an increasing number of dimensions. Interestingly, the gap between LCKD and kd-R grows slightly with dimensionality, due to the additional random cracks performed by kd-R. On a random query workload, such random cracks bring little benefit, while incurring an overhead that rises with dimensionality.



(a) CKD, QUASII, Static. **(b) Stochastic and hybrid.**

Figure 7: Effect of dimensionality.

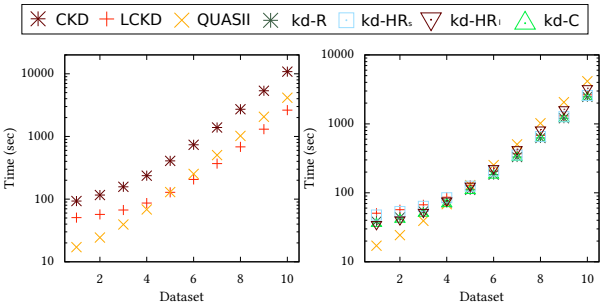
Henceforward, we use only LCKD and its stochastic and hybrid variants as representatives of cracking KD-trees and drop Static from figures for the sake of readability.



(a) Time per query **(b) Cumulative time**

Figure 8: Skyserver $^1/_4$, chronological order.

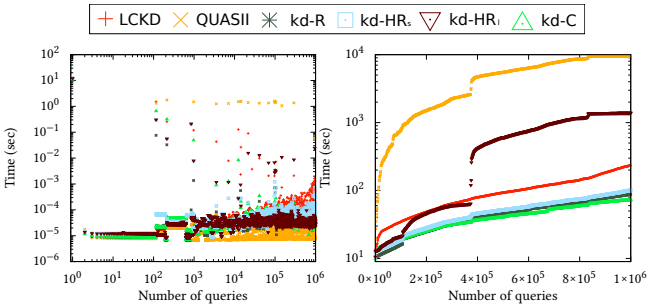
Skyserver workload, chronological. Now we apply the Skyserver chronologically ordered workload on Skyserver data. Figure 8 shows our results with the Skyserver $^1/_4$ dataset. In the first 500 queries, QUASII occasionally reaches response times comparable to the other methods, which should correspond to accessing already indexed data areas; progressively expensive queries become rarer, and eventually QUASII converges to response times lower than those of other methods. This converged response time of QUASII does not render its cumulative time lower than those of others for the entire workload in this configuration; kd-C and kd-R achieve the best cumulative times. Still, as Figure 9 shows, as we reduce the data size (10 for full, 1 for $^1/_{512}$) under the same workload so as to render the workload to data ratio larger, eventually QUASII becomes the fastest.



(a) QUASII, CKD, LCKD. **(b) Stochastic and Hybrid.**

Figure 9: Skyserver; regular workload, variable data sizes.

Figure 11 presents cumulative times with the same workload and four different Skyserver sizes. As data size falls, the time of QUASII approaches and supersedes, those of kd-HR, kd-R, and LCKD; still, kd-R and kd-C remain better options than QUASII for workloads reasonably large compared to the data.



(a) Time per query **(b) Cumulative time**

Figure 10: Skyserver $^1/_{64}$, sequential workload.

Skyserver workload, sequential. We now assess performance on the Skyserver 1/64 data against the sequential Skyserver workload, sorted by the dec celestial coordinate. Figure 10 presents our results. Despite the smaller data size compared to those we examined previously, the sequential nature of the workload bears upon QUASII, which presents the worst result in cumulative time. LCKD achieves better cumulative time than QUASII, yet its response time does not converge as well; the tree it builds grows progressively higher in a lopsided manner. The stochastic variants, kd-C and kd-R, eschew the deficiencies of query-driven methods and attain best performance, while kd-HR$_s$ follows suit. On the other hand, kd-HR$_\ell$, which resorts to QUASII quite early, inherits the liability of QUASII. This result reconfirms that the QUASII strategy is a liability more than an asset in hybrids.

Skyserver workload sorted by size Now we apply the Skyserver workload ordered by query size on the Skyserver 1/4 data. Figures 12 and 13 show the results for *ascending* and *descending* order, respectively. On the ascending order, as expected, QUASII is initially slower, but achieves better query response times later. In cumulative time, kd-C performs best, closely followed by kd-R; kd-HR$_s$ does not gain from its hybrid character, while kd-HR$_\ell$ has a clear disadvantage. LCKD is superseded by kd-Rand kd-HR. On

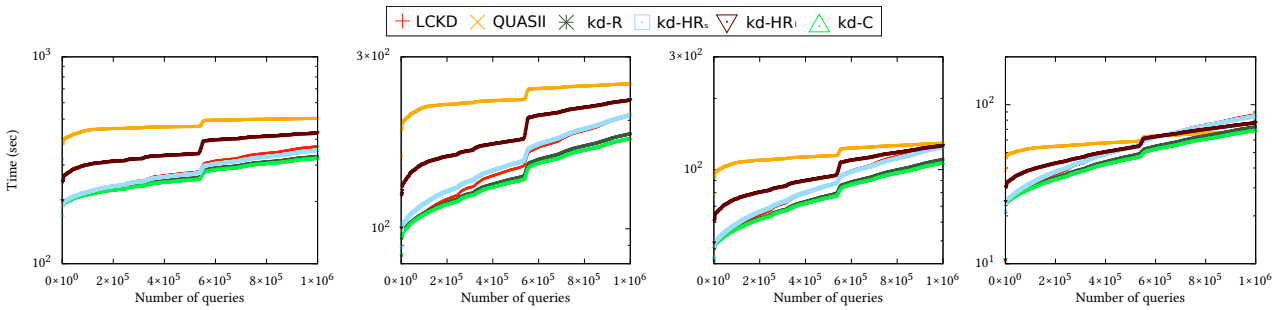
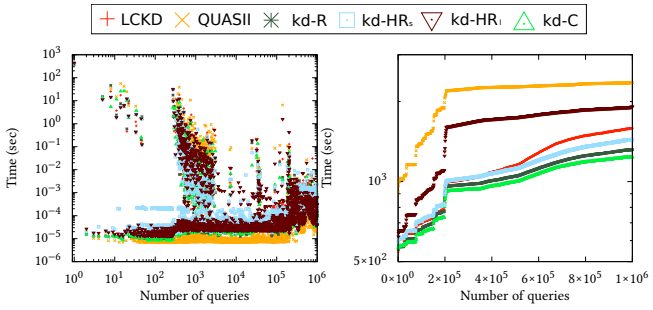


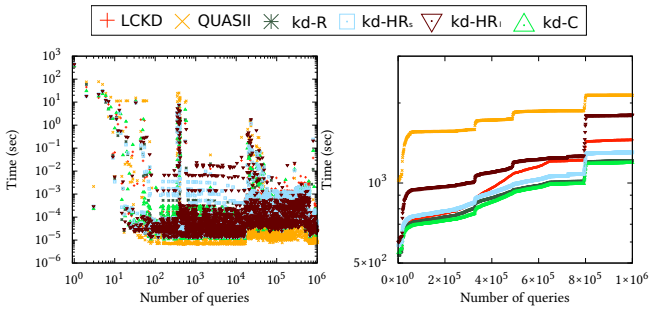
Figure 11: Skyserver, regular workload, datasets: 1/8, 1/16, 1/32, 1/64.

the descending order, time per query is initially higher, as queries of larger extent require more indexing work; cumulative time grows more steeply in the early stage than with the ascending order, yet performance resembles the ascending case.



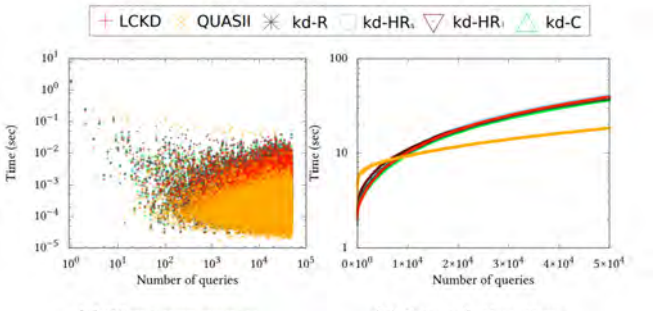
(a) Time per query **(b) Cumulative time**

Figure 12: Skyserver 1/2, workload sorted on size (asc.).



(a) Time per query **(b) Cumulative time**

Figure 13: Skyserver 1/2, workload sorted on size (desc.).



(a) Time per query **(b) Cumulative time**

Figure 14: Neuroscience data, random cluster workload.

Neuroscience data. We now assess performance on the Neuroscience data with the random clustered workload. This dataset contains objects with spatial extent rather than points; thus, we employ *query window extension* [15], as in [12]: we represent shapes by their lower coordinates per dimension and *extend* query ranges by the maximum object extent towards the lower side of each dimension, to hit the lower coordinates of any object overlapping the query's range; we filter *false hits* in a refinement step. Figure 14 presents our results. No method converges as robustly as with point data, due to the overhead caused by query extension. Still, QUASII converges more robustly than others.

6 CONCLUSION

We conducted a comparative experimental evaluation of works on multidimensional adaptive indexing and enhancements leveraging stochastic and hybrid strategies. We found that adaptations of the Cracking KD-tree achieve better performance compared to QUASII in terms of initialization and with short workloads, while QUASII yields attractive performance with long-running workloads. We combined the Cracking KD-Tree with stochastic measures that ameliorate the sensitivity to the order in which queries are posed. Further research is needed on multidimensional adaptive indexing of objects with spatial extent and accommodating updates; we also aim to investigate the adaptive indexing of graph structures with privacy constraints [11, 17] and adaptive multidimensional synopses [9].

REFERENCES

[1] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.
[2] Goetz Graefe and Harumi A. Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*. 371–381.
[3] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.
[4] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB* 5, 6 (2012), 502–513.
[5] Pedro Holanda, Matheus Nerone, Eduardo Cunha de Almeida, and Stefan Manegold. 2018. Cracking KD-Tree: The First Multidimensional Adaptive Indexing (Position Paper). In *DATA*. 393–399.
[6] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*. 68–78.
[7] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2009. Self-organizing tuple reconstruction in column stores. In *SIGMOD*. 297–308.
[8] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB* 4, 9 (2011), 585–597.
[9] Panagiotis Karras and Nikos Mamoulis. 2008. Hierarchical synopses with optimal error guarantees. *ACM Trans. Database Syst.* 33, 3 (2008), 18:1–18:53.
[10] Panagiotis Karras, Artyom Nikitin, Muhammad Saad, Rudrika Bhatt, Denis Antyukhov, and Stratos Idreos. 2016. Adaptive Indexing over Encrypted Numeric Data. In *SIGMOD*. 171–183.
[11] Sadegh Nobari, Panagiotis Karras, HweeHwa Pang, and Stéphane Bressan. 2014. L-opacity: Linkage-Aware Graph Anonymization. In *EDBT*. 583–594.
[12] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2018. QUASII: QUery-Aware Spatial Incremental Index. In *EDBT*. 325–336.
[13] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Uncracked Pieces in Database Cracking. *PVLDB* 7, 2 (2013), 97–108.
[14] DR16 Sloan Digital Sky Survey. 2018. http://cas.sdss.org/.
[15] Emmanuel Stefanakis, Yannis Theodoridis, Timos K. Sellis, and Yuk-Cheung Lee. 1997. Point Representation of Spatial Objects and Query Window Extension: A New Technique for Spatial Access Methods. *IJGIS* 11, 6 (1997).
[16] Yufei Tao and Dimitris Papadias. 2002. Adaptive Index Structures. In *VLDB*.
[17] Mingqiang Xue, Panagiotis Karras, Chedy Raïssi, Panos Kalnis, and Hung Keng Pung. 2012. Delineating Social Network Data Anonymization via Random Edge Perturbation. In *CIKM*. 475–484.
[18] Fatemeh Zardbani, Peyman Afshani, and Panagiotis Karras. 2020. Revisiting the Theory and Practice of Database Cracking. In *EDBT*. 415–418.

Short Paper

Twin Subsequence Search in Time Series

Georgios Chatzigeorgakidis
IMSI, Athena R.C.
gchatzi@athenarc.gr

Dimitrios Skoutas
IMSI, Athena R.C.
dskoutas@athenarc.gr

Kostas Patroumpas
IMSI, Athena R.C.
kpatro@athenarc.gr

Themis Palpanas
LIPADE, Université de Paris &
French University Institute (IUF)
themis@mi.parisdescartes.fr

Spiros Athanasiou
IMSI, Athena R.C.
spathan@athenarc.gr

Spiros Skiadopoulos
DIT, University of Peloponnese
spiros@uop.gr

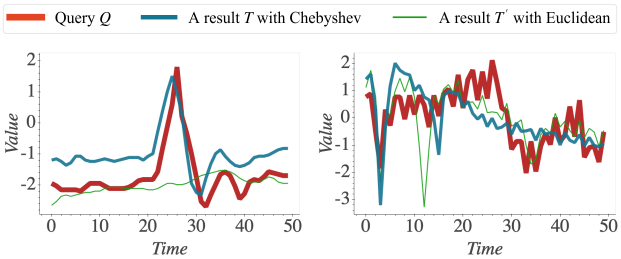
ABSTRACT

We address the problem of subsequence search in time series using Chebyshev distance, to which we refer as twin subsequence search. We first show how existing time series indices can be extended to perform twin subsequence search. Then, we introduce TS-Index, a novel index tailored to this problem. Our experimental evaluation compares these approaches against real time series datasets, and demonstrates that TS-Index can retrieve twin subsequences much faster under various query conditions.

1 INTRODUCTION

Given a time series T and a query sequence Q ($|Q| \ll |T|$), *subsequence search* finds subsequences in T that are similar to Q. Although most works rely on Euclidean distance or Dynamic Time Warping (DTW) (e.g., [17, 19]), different $\mathcal{L}_p$ norms or other similarity measures are also useful for capturing different patterns of similarity or achieving higher classification accuracy in certain datasets [6, 22]. In this work, we use the *Chebyshev* distance (i.e., $\mathcal{L}_\infty$ norm) between two subsequences, which is the maximum difference of their values across their entire duration. We call two subsequences *twins* with respect to a distance threshold ϵ, if their Chebyshev distance is not greater than ϵ. This kind of similarity search can be useful is various applications: finding *doublet* earthquakes in seismology, identifying similar traffic patterns in road networks, or detecting irregular patterns in medical applications like Electroencephalography (EEG) or Electrocardiography (ECG) sequences, etc.

The following indicative experiment on an EEG time series [12] with length of 1,801,999 timestamps provides some insight on the different results obtained using Chebyshev distance as opposed to Euclidean. Considering a query sequence Q and a Chebyshev distance threshold ϵ, we identify all twin subsequences, obtaining 1,034 results in total. We then attempt to retrieve the same results by subsequence search using Euclidean distance. To avoid any false negatives, as will be shown later in Section 3.1, we need to set the Euclidean distance threshold to $\epsilon' = \epsilon \times \sqrt{|Q|}$. The latter produces 127,887 results. Figure 1 exemplifies the intuition behind matches obtained with Chebyshev distance compared to those with Euclidean, for two different queries. Assume a query sequence Q and two matches, T and T', obtained under Chebyshev and Euclidean distance, respectively. As shown, T closely matches the query in all timestamps. Instead, T' either lacks a spike that is present in the query (Fig. 1a) or exhibits one that is not present in the query (Fig. 1b).



(a) Absence of desired spike (b) Presence of undesired spike

Figure 1: Examples of false positives obtained with Euclidean distance compared to results with Chebyshev distance on subsequences of the EEG dataset.

Given a query sequence Q and a time series T, a naïve process for finding twin subsequences of Q across T is by performing a sweepline scan. This scans T using a sliding window of length $|Q|$, comparing at each timestamp the query with the current subsequence extracted from T, and adding it to the results if it satisfies the given threshold ϵ on Chebyshev distance. However, this is clearly inefficient for long time series.

In this work, we investigate index-based methods for twin subsequence search. First, we show how two state-of-the-art time series indices, namely KV-Index [19] and *i*SAX [18] can be adapted for this task. Then, we introduce a novel index, called TS-Index, which is tailored to this problem. TS-Index is a tree structure that summarizes the subsequences contained within each node using *Minimum Bounding Time Series* (MBTS) [4], consisting of an upper and lower bounding sequence. Our experimental evaluation shows that executing twin subsequence search using TS-Index is significantly faster compared to adapting the query execution over other indices.

Specifically, our main contributions are as follows:

- We introduce the problem of twin subsequence search and propose a filter-verification algorithm that can be applied on state-of-the-art time series indices.
- We then introduce TS-Index, a tree-based index tailored to twin subsequence search, which utilizes appropriate bounds in its nodes to prune the search space.
- We experimentally evaluate our proposed methods using real-world datasets in terms of query execution, memory footprint and index construction time.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 formally defines the problem. Section 4 presents how it can be addressed based on existing indices. Section 5 presents the proposed TS-Index. Section 6 reports our experimental results. Finally, Section 7 concludes the paper.

2 RELATED WORK

Subsequence search can be performed with a sweepline approach that scans the time series using a sliding window. Various optimizations can be found in UCR suite [17] and Matrix Profile [21]. However, these optimizations are specific to Euclidean distance and thus cannot be applied to twin subsequences. Also, the lack of an index poses efficiency and scalability limitations.

A survey of time series indices for similarity search can be found in [7]. Several methods use Discrete Wavelet Transform to reduce dimensionality and then generate an index based on the transformed sequences (e.g., [3, 16]). More recent approaches are based on the *Symbolic Aggregate Approximation* (SAX) representation of time series [10]. A SAX *word* is a multi-resolution summary of a time series quantized on the value domain. It is derived from the Piecewise Aggregate Approximation (PAA) [8], which segments a time series on the time axis and approximates it by retaining only the mean value per segment. This has led to the *i*SAX index [18], a tree-based structure built over the SAX words of a set of time series. Each node in *i*SAX contains a SAX word that guarantees a lower bound in terms of Euclidean distance for all the time series indexed by it. To answer similarity search queries, the index is traversed in a top-down fashion, comparing at each step the SAX representation of the query against the ones contained in each visited node. Several extensions to *i*SAX have been proposed [13]. *i*SAX 2.0 [1] and *i*SAX2+ [2] enable bulk loading, while ADS+ [23] builds the index adaptively, based on the query workload. DP*i*SAX [20] is a distributed index. ParIS [14] and MESSI [15] take advantage of modern multi-core architectures. Coconut [9] introduces sortable SAX representations and builds an index in a bottom-up fashion. Finally, ULISSE [11] answers queries of varying length.

Another recent method for subsequence search is KV-Index [19]. After extracting all subsequences of a given length from a time series and deriving their corresponding mean values, it generates an index containing key-value pairs. Each key represents a range of mean values for a group of subsequences, pointing to starting positions of these subsequences along the original time series.

As we show in Section 4, it is possible to execute twin subsequence search queries using *i*SAX or KV-Index. However, since these indices are tailored to similarity search using Euclidean distance, this approach is suboptimal, as indicated also in our experiments in Section 6.

An index for arbitrary $\mathcal{L}_p$ norms is described in [22]. It divides each sequence into a fixed number of equi-sized segments, and takes the mean of each segment to form a feature vector. Such a generic approach favors flexibility; instead, our focus in this paper is on optimizing performance specifically for queries using Chebyshev distance.

Finally, in a previous work [5], we have studied the problem of discovering pairs and bundles of similar time-aligned subsequences within a collection of time series, based on Chebyshev distance, using a sweepline approach. In this paper, we focus on searching for twin subsequences in an input time series T that are similar to a query subsequence Q, which is a different problem, and we propose an index-based approach. Furthermore, in another previous work [4], we have developed a hybrid index, called BTSR-Tree, which also employs the concept of Minimum Bounding Time Series (MBTS) to prune the search space. However, this is a spatial-first index specifically tailored to queries over geo-located time series, and it is based on Euclidean distance instead of Chebyshev.

3 PROBLEM DEFINITION

Next, we formally introduce the problem of twin subsequence search and describe a generic filter-verification approach.

3.1 Problem Statement

A *time series* is a time-ordered sequence $T = \{T_1, T_2, ..., T_n\}$, where T_i is the value at the i-th timestamp and $n = |T|$ is the length of the series (i.e., number of timestamps). We use $T_{p,l}$ to denote the *subsequence* $\{T_p, ..., T_{p+l-1}\}$ starting at timestamp p and having length l, where $1 \leq p \leq p + l - 1 \leq n$. For brevity, we also use S to generally refer to a (sub)sequence.

Given two sequences S and S' of equal length l, we call them *twins* if their Chebyshev distance is not greater than a given threshold ϵ. The Chebyshev distance of two vectors is their maximum difference along any dimension. Hence, if S and S' are twin sequences with respect to ϵ, their values at any timestamp should not differ by more than ϵ. Formally:

Definition 1 (Twin Sequences). *Two sequences S and S' of equal length l are called* twins *with respect to a given threshold ϵ, denoted as $S_1 \sim_\epsilon S_2$, if their Chebyshev distance d is not greater than ϵ, i.e., $d(S, S') := \max_{i=0}^{l-1}(|S_i - S'_i|) \leq \epsilon$.*

We can now formally define the problem:

Problem 1 (Twin Subsequence Search). *Given a query sequence Q of length l, a time series T of length $n \gg l$, and a distance threshold ϵ, find all subsequences S in T ($|S| = l$) such that $Q \sim_\epsilon S$.*

We note two important observations below. Given two twin sequences $S \sim_\epsilon S'$ of length l, their Euclidean distance is $ED(S, S') = \sqrt{\sum_i (S_i - S'_i)^2} \leq \sqrt{\sum_i \epsilon^2} = \epsilon \times \sqrt{l}$. This establishes a relation between a given Chebyshev distance threshold and a corresponding Euclidean distance threshold. Moreover, from Definition 1, it follows that any pair of time-aligned subsequences across two twin sequences are also twins, i.e., if $T \sim_\epsilon T'$, then $T_{p,l} \sim_\epsilon T'_{p,l}$ for any $l \in [1, |T|]$ and $p \in [1, |T| - l]$.

Often, z-normalization is applied when comparing time series. Throughout the paper, we consider various possibilities: (a) working with the raw values, (b) z-normalizing the entire time series, (c) z-normalizing each individual subsequence. We discuss the implications of each case where relevant.

3.2 Filter-Verification Approach

We can detect twin subsequences following a filter-verification framework: the first step (*filtering*) generates candidate subsequences, which are then evaluated in the second step (*verification*) to identify those satisfying the Chebyshev distance threshold. A straightforward approach for generating candidates is to scan the entire time series T with a *sweepline* and consider each subsequence $T_{p,l}$ for $p \in [1, |T| - l]$ as a candidate.

Verification is done by checking all pairwise value differences between Q and $T_{p,l}$. If the difference found at a timestamp exceeds ϵ, then candidate $T_{p,l}$ is rejected, otherwise it is accepted. Verification can be accelerated by detecting false positives as early as possible. If the values are z-normalized, we can prioritize those points in Q having the highest absolute value, since these are less likely to have a match with the respective points in $T_{p,l}$. This optimization is also used in *UCR Suite* [17], and is known as *reordering early abandoning*.

The drawback of this sweepline approach is that it generates an excessive number of candidates (specifically, $|T| - l$), thus

incurring a prohibitive cost when dealing with long series. To filter candidates more effectively, in the following sections we present methods based on indexing the subsequences of T. First, we address the problem using state-of-the-art indices; then, we introduce a novel index tailored to twin subsequence search.

4 TWIN SUBSEQUENCE SEARCH WITH EXISTING INDICES

Next, we focus on two representative state-of-the-art indices for time series similarity search, namely KV-Index [19] and *i*SAX [2], showing how they can be used for twin subsequence search without altering their structure.

4.1 KV-Index

Given a time series T, KV-Index [19] is built by considering all its subsequences of a pre-defined length l. Each subsequence S is represented by a pair (p, μ), where p is its starting position (i.e., timestamp) in T and μ is its mean value over the next l timestamps. KV-Index is an inverted index constructed over these pairs. Each key is a range of mean values, whereas each inverted list entry contains intervals of positions.

Twin subsequence search can be performed with KV-Index based on the following observation. If two subsequences S and S' of length l are twins with respect to ϵ, i.e., $S \sim_\epsilon S'$, then their mean values μ and μ' cannot differ by more than ϵ, i.e., $|\mu - \mu'| \leq \epsilon$. Based on this, we can use a KV-Index built over a time series T to generate candidates for detecting twin subsequences. Specifically, assume a query sequence Q with mean value μ_q. The candidate subsequences in T are those included in the inverted lists with keys $[\mu_{min}, \mu_{max}]$, such that $\mu_{min} - \epsilon \leq \mu_q \leq \mu_{max} + \epsilon$. Then, the obtained candidates must be verified to derive the final results. Notice that this property is not effective if each individual subsequence has been z-normalized, because then all mean values are zero. Hence, KV-Index is applicable when working with raw values or if the entire sequence is z-normalized.

4.2 *i*SAX Index

*i*SAX is a tree index structure for time series similarity search [2]. Time series are z-normalized and indexed using their *Symbolic Aggregate approXimation (SAX)* [18]. The SAX representation of a series is derived in two steps. The first applies *Piecewise Aggregate Approximation* (PAA) [8], which splits the series in a specified number m of segments and approximates each one with the mean value over the corresponding time interval. The second step applies quantization to assign each mean value to a discrete SAX *symbol*. Hence, each SAX symbol X corresponds to a range of mean values $[\mu_{X_{min}}, \mu_{X_{max}})$. The SAX representation of a series is a sequence of m SAX symbols (one symbol per segment), and is called SAX *word*. Notice that, by default, SAX words are derived using precomputed breakpoints that are selected assuming z-normalized values; nevertheless, non-normalized values can also be handled by adjusting the breakpoints accordingly.

Twin subsequence search can be enabled over *i*SAX by reasoning as follows. Assume two subsequences S and S' of length l, and their SAX representations $SAX(S) = \{X_1, X_2, ..., X_m\}$ and $SAX(S') = \{X'_1, X'_2, ..., X'_m\}$. As we have observed earlier, (a) if two sequences are twins with respect to a threshold ϵ, then the difference between their mean values is also bounded by ϵ, and (b) any pair of time-aligned segments across two twin sequences are also twins. Combining these two properties, we can see that if $S \sim_\epsilon S'$, then for each pair of symbols X_i and X'_i in the respective SAX representations, the mean values denoted by these symbols must not differ by more than ϵ. Hence, if $S \sim_\epsilon S'$, then $\mu_{X_{i\,max}} \geq \mu_{X'_{i\,min}} - \epsilon$ and $\mu_{X_{i\,min}} \leq \mu_{X'_{i\,max}} + \epsilon$ for any $i \in [1, m]$.

Consequently, we can perform twin subsequence search using *i*SAX as follows. Given a time series T, we construct an *i*SAX index over all its l-length subsequences. Then, for a query sequence Q, we traverse the *i*SAX index starting from its root. At each node, we check the SAX word of Q against the SAX word of that node, applying the property mentioned above. If the check fails, the node and its subtree can be safely pruned; otherwise, the search continues at the node's children. Once a leaf node is reached, and qualifies according to this check, all subsequences indexed therein are retrieved as candidates for verification.

5 THE TS-INDEX

As discussed in Section 4, it is possible to use KV-Index or *i*SAX to identify candidates for twin subsequence queries. However, since these indices are not tailored to the matching criterion, they tend to generate a large number of false positives, incurring a significant verification cost, as confirmed in our experiments. In the following, we introduce TS-Index, which is specifically designed for twin subsequence search. First, we provide an overview of its structure and explain how it is constructed. Then, we present an algorithm to evaluate twin subsequence queries specifying a distance threshold.

5.1 Index Structure

The core concept in TS-Index is that of *Minimum Bounding Time Series* (MBTS) [4]. An MBTS is a pair of sequences that fully encloses a set of time series $\mathcal{T}$ by indicating the maximum and minimum values at each timestamp. Figure 2a depicts an example of an MBTS enclosing a set of four time series. Formally:

Definition 2 (MBTS). *Given a set $\mathcal{T}$ of time series with equal length l, its MBTS $B = (B^{\sqcap}, B^{\sqcup})$ consists of an* upper bounding time series $B^{\sqcap}$ *and a* lower bounding time series $B^{\sqcup}$, *constructed by respectively selecting the maximum and minimum values at each timestamp $i \in \{1, \ldots, l\}$ among all time series in $\mathcal{T}$ as follows:*

$$B^{\sqcap} = \{\max_{T \in \mathcal{T}} T_1, \ldots, \max_{T \in \mathcal{T}} T_l\}$$
$$B^{\sqcup} = \{\min_{T \in \mathcal{T}} T_1, \ldots, \min_{T \in \mathcal{T}} T_l\} \quad (1)$$

The TS-Index has a tree structure. Each internal node points to a set of children nodes, whereas each leaf node points to a set of subsequences (more specifically, to the starting positions of its indexed subsequences along the input time series T). All leaf nodes are at the same level. Each node is associated with an MBTS, which encloses all the sequences indexed therein. Clearly, MBTS get tighter when descending from the root to the leaf level. Figure 3a illustrates an example of TS-Index for nine input sequences. The MBTS of each node is depicted as a grey band.

5.2 Index Construction

Assume an input time series T and a subsequence length l. The TS-Index over T is constructed in a top-down fashion, by sequentially inserting all l-length subsequences of T. When inserting a sequence S, we traverse the index from the root, selecting at each level the node whose MBTS has the smallest distance from S, until a leaf node is reached. The distance between a sequence S and an MBTS B is calculated using the following formula:

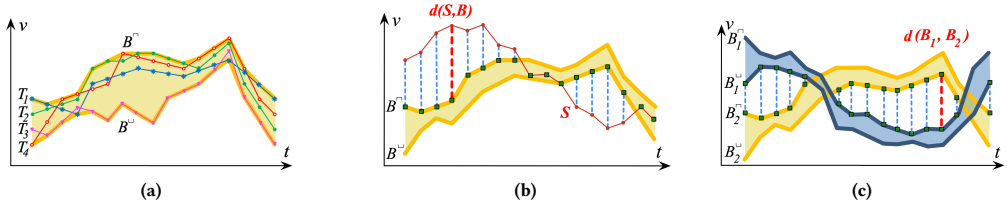


Figure 2: (a) MBTS enclosing a set of 4 time series. Distance between (b) a sequence S and an MBTS B, (c) MBTS B_1 and B_2.

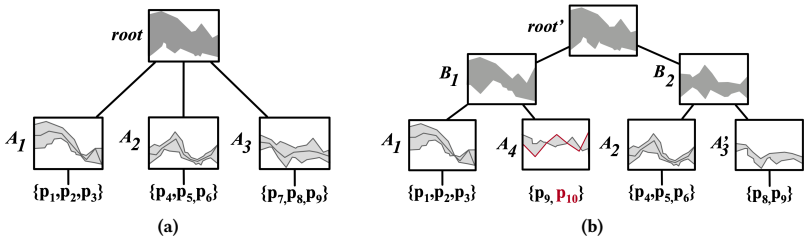


Figure 3: (a) TS-Index for 9 input sequences. (b) Inserting p_{10} causes a split at leaf A_3 and splits propagate upwards.

$$d(S,B) = \max_i \begin{cases} S_i - B_i^{\sqcap} & \text{if } S_i > B_i^{\sqcap} \\ B_i^{\sqcup} - S_i & \text{if } S_i < B_i^{\sqcup} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

where $B_i^{\sqcap}$ and $B_i^{\sqcup}$ are the i^{th} values of the upper and lower bounds of the MBTS B, respectively.

Each node has a minimum capacity μ_c and a maximum *capacity* M_c, specifying the minimum and maximum number of children it can point to. Once a node exceeds M_c, it is split in two nodes. This may cause the parent node to also exceed the maximum capacity M_c, in which case it is split too. Hence, this process recursively propagates upwards until no further splits occur. This procedure ensures that all leaves are placed on the same level of the tree.

During node splitting, the goal is to make the MBTS of each new sibling node as tight as possible. If this is a leaf node, we identify the two subsequences within the original node having the highest Chebyshev distance and use them as seeds for the two sibling nodes. Each remaining subsequence is assigned to the node where it causes the smallest expansion of its MBTS, which gets updated accordingly. For an internal node, the process is similar. Yet, adjusting its MBTS in this case involves the MBTS of children nodes instead of individual sequences. To accommodate this, the distance between two MBTS B_1 and B_2 is defined as:

$$d(B_1,B_2) = \max_i \begin{cases} B_{1,i}^{\sqcup} - B_{2,i}^{\sqcap} & \text{if } B_{1,i}^{\sqcup} > B_{2,i}^{\sqcap} \\ B_{2,i}^{\sqcup} - B_{1,i}^{\sqcap} & \text{if } B_{1,i}^{\sqcap} < B_{2,i}^{\sqcup} \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

where $B_{1,i}^{\sqcup}$, $B_{1,i}^{\sqcap}$ and $B_{2,i}^{\sqcup}$, $B_{2,i}^{\sqcap}$ are the i^{th} values of the upper and lower bounds of the MBTS B_1 and B_2, respectively. Figures 2b and 2c exemplify the calculation of the distance of a sequence S to an MBTS B and the calculation of the distance between two MBTS (B_1, B_2) respectively; in both cases, the distance is the length of the dashed red line.

Figure 3b depicts an example where inserting subsequence p_{10} into leaf node A_3 of the TS-Index in Figure 3a, causes it to split into two new nodes, A_3' and A_4 (we assume $\mu_c = 2$ and M_c=3). This process is then propagated upwards, splitting the root into B_1 and B_2. To keep the MBTS tight –according to Equation 3–, nodes A_1, A_4 have become children of B_1 and A_2, A_3' are now children of B_2. Finally, a new root is added, increasing the index height by one.

5.3 Query Execution

Twin subsequence search can be performed with TS-Index based on the following lemma.

Lemma 1. *Assume a query sequence Q and a node N of the TS-Index with MBTS B. If there exists a sequence S indexed at N such that $Q \sim_\epsilon S$, then $d(Q,B) \leq \epsilon$.*

Proof. Assume that $Q \sim_\epsilon S$ for a sequence S indexed by node N. From Definition 2, it follows that $S_i \in [B_i^{\sqcup}, B_i^{\sqcap}]$ for each timestamp i. Moreover, from Definition 1, it follows that $|Q_i - S_i| \leq \epsilon$. Hence, from Equation 2, we derive $d(Q,B) \leq \epsilon$. □

Given a query sequence Q, we traverse the index in a top-down fashion, starting from its root. For each visited node N, we compare Q against N's MBTS, applying Lemma 1 to prune its subtree. Note that this check can be accelerated, since it is not necessary to fully compute distance $d(Q,B)$; instead, if the indexed values have been z-normalized, we apply early abandoning (see Section 3.2) to prune the node as soon as the value difference exceeds ϵ in at least one timestamp. Multiple paths starting from the root may need to be explored, depending on the query and the tightness of the bounds in the visited nodes.

Algorithm 1 describes the search process. The input includes the query sequence Q, the constructed TS-Index I, the given time series T and the threshold ϵ. We start by initializing a list L with the root's children (Line 2). Then, we traverse the index by iterating over this list (Lines 3-12). For each node N currently in the list, we obtain its MBTS (Lines 4-5). Then, we check whether the distance between this MBTS and the query is higher than the specified threshold ϵ (Line 6). If so, the subtree under the current node N is pruned; otherwise, it is examined as explained next. If N is not a leaf node, we insert its children in list L for probing (Lines 7-8). Once a leaf node is reached, we iterate over all the subsequence positions it contains and check whether each corresponding subsequence is a twin of Q with respect to ϵ. If so, we add this subsequence to the final results (Lines 9-12). The results are returned once all candidate nodes in list L have been either probed or pruned (Line 13).

Algorithm 1: `TwinSubsequenceSearch`

Input : Time series T, TS-Index I, query Q, threshold ϵ
Output: List R of twin subsequences to Q

```
1  R ← ∅
2  L ← I.root.getChildren()
3  while L ≠ ∅ do
4      N ← L.getNext()
5      B ← N.MBTS
6      if d((Q, B) ≤ ε then
7          if N is not leaf then
8              L ← L ∪ {N.getChildren()}
9          else
10             foreach p ∈ N.getPositions() do
11                 if d((Q, T_{p,l}) ≤ ε then
12                     R ← R ∪ T_{p,l}
13 return R
```

Table 1: Datasets and distance thresholds.

Dataset	n	ϵ (norm)	ϵ (non-norm)
Insect	64,436	0.5,**0.75**,1,1.25,1.5	50, **100**,150,200,250
EEG	1,801,999	0.1,0.2,**0.3**,0.4,0.5	20, **40**,60,80,100

Table 2: Other parameters.

Parameter	*Value*
Number m of segments	5, **10**, 20, 25, 50
Sequence length l	50, **100**, 150, 200, 250

6 EXPERIMENTAL EVALUATION

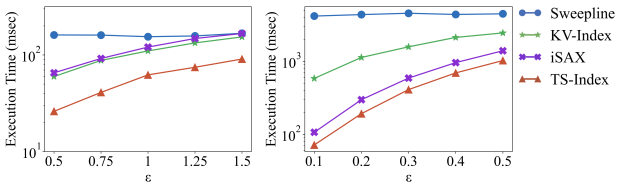
Next, we present an experimental evaluation of our methods against two real-world datasets.

6.1 Experimental Setup

We performed experiments against two real-world time series (see Table 1), which contain diverse patterns and differ in their total duration. In particular, the *Insect Movement* [12] series contains 64,436 insect telemetry readings spanning around 30 minutes (36 readings/sec), whereas the *Electroencephalography (EEG)* [12] series comprises 1,801,999 EEG readings at 500Hz lasting one hour. Unless stated otherwise, we z-normalize the time series to facilitate selection of distance thresholds.

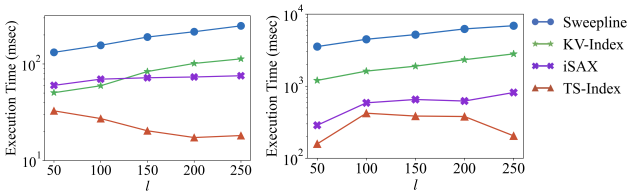
Table 1 indicates the different values for the distance threshold ϵ used in the experiments against each dataset, for z-normalized (*norm*) or original values (*non-norm*). Table 2 contains the values for subsequence length l and number of segments m, which are common in the experiments on both datasets. In both tables, default values are in bold. These values have been selected after running several preliminary tests, which also guided selection of other parameters. Specifically, for *i*SAX, the maximum node capacity is set to 10,000 to enable index construction in reasonable time even for larger datasets. The default values for minimum and maximum node capacity in TS-Index are set to $\mu_c = 10$ and $M_c = 30$, respectively.

For each dataset, we randomly picked 100 subsequences, each of length $l = 100$ points, and used them as the query workload in all tests against that dataset. We report average response time per query (in milliseconds). We implemented all methods, including KV-Index, *i*SAX, and TS-Index, in Java. In all implementations, the structure of the index is kept in memory, while the original input dataset is stored on disk. Leaf nodes in the index contain the starting positions of the subsequences in the input time series. Thus, when a leaf is reached at query time, its corresponding subsequences are obtained from the input time series file using random access. All experiments were conducted on a server with 4 CPUs, each equipped with 8 cores clocked at 2.13GHz, and 256 GB RAM running Debian Linux.



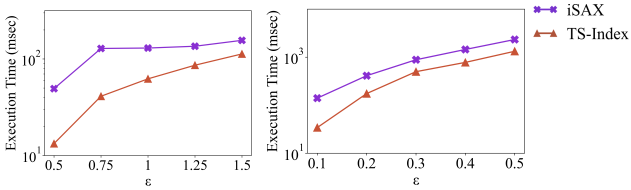
(a) Insect Dataset (b) EEG Dataset

Figure 4: Varying distance threshold ϵ.



(a) Insect Dataset (b) EEG Dataset

Figure 5: Varying subsequence length l.



(a) Insect Dataset (b) EEG Dataset

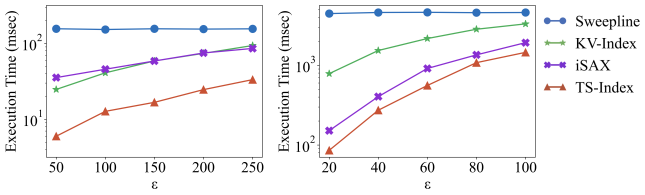
Figure 6: Varying ϵ on z-normalized subsequences.

6.2 Performance

We compare the average execution time per query for varying values of each parameter, setting the rest to their default values.

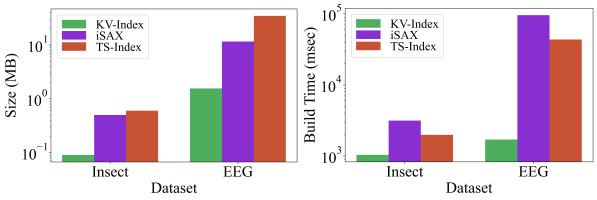
6.2.1 Varying threshold ϵ. Figure 4 depicts query execution time (in logarithmic scale) for varying threshold ϵ. As expected, searching with the Sweepline approach has a fixed cost per dataset regardless of ϵ, since it needs to scan all subsequences extracted from the input time series. Relaxing the threshold incurs an overhead when an index is involved. Queries against KV-Index perform poorly compared to other indices, since filtering based on mean values achieves less pruning. Searching with TS-Index outperforms the rest in every setting for both tested datasets. Overall, TS-Index is at least an order of magnitude more efficient in twin subsequence search compared to the KV-Index and Sweepline approaches. It is also consistently better than *i*SAX as it is less susceptible to fluctuations in the input sequences.

6.2.2 Varying Subsequence Length. Figure 5 plots performance results with a varying length l for subsequences obtained from the input time series. Increasing l seems to slightly negatively affect all approaches, except for TS-Index. Since longer subsequences are extracted, more checks are required, both in nodes (in case of *i*SAX) and raw subsequences during verification. Instead, TS-Index is faster when longer subsequences are specified, as it becomes less likely to find matching twins. In particular, TS-Index has higher pruning capability and can skip non-qualifying subtrees earlier at higher levels in the tree hierarchy. Thus, fewer



(a) Insect Dataset (b) EEG Dataset

Figure 7: Varying ϵ on non-normalized data.



(a) Memory Footprint (b) Build Time

Figure 8: Memory footprint and build time per index.

leaf nodes are accessed and need to be verified, saving much of the verification cost for checks per timestamp.

6.2.3 Searching over z-normalized subsequences. We repeat the experiment for varying distance threshold ϵ, this time applying z-normalization over each individual subsequence, before inserting it in the index. As mentioned in Section 4.1, KV-Index cannot be built on such data since the mean value per subsequence would always be zero; thus, we only compare TS-Index with *i*SAX. The results are depicted in Figure 6. Clearly, z-normalizing the subsequences separately has no significant effect on the performance of TS-Index; the results are similar to those in Figure 4, with TS-Index outperforming *i*SAX in all cases.

6.2.4 Searching on Non-Normalized Data. Query execution cost for identifying twin subsequences against the raw (non-normalized) time series is depicted in Figure 7. Overall, TS-Index copes better than all the rest even for raw data, confirming its suitability for twin subsequence search in various settings.

6.2.5 Index Size. Figure 8a presents the memory footprint of TS-Index, *i*SAX and KV-Index for each dataset. KV-Index requires less space than TS-Index and *i*SAX, as it only keeps in memory the mean value and position range per subsequence. Instead, TS-Index and *i*SAX occupy more space due to their more complex structures. Specifically, *i*SAX requires two to three times less space than TS-Index. Indeed, *i*SAX needs to store one SAX word per node, whereas a node in TS-Index is represented by an MBTS, hence its increased memory footprint. Nevertheless, all indices, including TS-Index, have sizes that easily fit in main memory.

6.2.6 Build Time. Similarly to the index size, and due to the significantly less required calculations (i.e., only subsequence mean values need be calculated and no node splitting is needed), KV-Index requires significantly less time to be constructed than *i*SAX and TS-Index (Figure 8b). *i*SAX is the slowest index to be built, since it needs to additionally convert the PAA of each subsequence to a SAX word for each extracted subsequence.

7 CONCLUSIONS

In this paper, we have introduced the twin subsequence search problem. Given a query sequence Q, an input time series T and a distance threshold ϵ, this task retrieves all subsequences in T with Chebyshev distance to Q not higher than ϵ. To answer this query efficiently, we have introduced the TS-Index. We have described the index structure and proposed algorithms for efficient index construction and query answering. Our experimental evaluation assesses the TS-Index in terms of construction cost and confirms its superiority for twin subsequence search queries when compared to the state-of-the-art.

ACKNOWLEDGMENTS

This work was supported by the EU H2020 project SmartDataLake (825041), the EU H2020 project OpertusMundi (870228) and the NSRF 2014-2020 project HELIX (5002781).

REFERENCES

[1] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn J. Keogh. 2010. iSAX 2.0: Indexing and Mining One Billion Time Series. In *ICDM*. 58–67.
[2] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn J. Keogh. 2014. Beyond one billion time series: indexing and mining very large time series collections with i SAX2+. *Knowl. Inf. Syst.* 39, 1 (2014), 123–151.
[3] Kin-pong Chan and Ada Wai-Chee Fu. 1999. Efficient Time Series Matching by Wavelets. In *ICDE*. 126–133.
[4] Georgios Chatzigeorgakidis, Dimitrios Skoutas, Kostas Patroumpas, Spiros Athanasiou, and Spiros Skiadopoulos. 2017. Indexing Geolocated Time Series Data. In *SIGSPATIAL*. 19:1–19:10.
[5] Georgios Chatzigeorgakidis, Dimitrios Skoutas, Kostas Patroumpas, Themis Palpanas, Spiros Athanasiou, and Spiros Skiadopoulos. 2019. Local pair and bundle discovery over co-evolving time series. In *SSTD*. 160–169.
[6] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. 2008. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1542–1552.
[7] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2018. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB* 12, 2 (2018), 112–127.
[8] Eamonn J. Keogh, Kaushik Chakrabarti, Michael J. Pazzani, and Sharad Mehrotra. 2001. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *Knowl. Inf. Syst.* 3, 3 (2001), 263–286.
[9] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A scalable bottom-up approach for building data series indexes. *PVLDB* 11, 6 (2018), 677–690.
[10] Jessica Lin, Eamonn J. Keogh, Li Wei, and Stefano Lonardi. 2007. Experiencing SAX: a novel symbolic representation of time series. *Data Min. Knowl. Discov.* 15, 2 (2007), 107–144.
[11] Michele Linardi and Themis Palpanas. 2020. Scalable Data Series Subsequence Matching with ULISSE. *VLDBJ* 11, 13 (2020), 2236–2248.
[12] Abdullah Mueen, Eamonn Keogh, Qiang Zhu, Sydney Cash, and Brandon Westover. 2009. Exact discovery of time series motifs. In *SIAM*. 473–484.
[13] Themis Palpanas. 2020. Evolution of a Data Series Index. *CCIS* 1197 (2020).
[14] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2018. ParIS: The Next Destination for Fast Data Series Indexing and Query Answering. In *IEEE BigData*.
[15] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2020. MESSI: In-Memory Data Series Indexing. In *ICDE*. 337–348.
[16] Ivan Popivanov and Renée J. Miller. 2002. Similarity Search Over Time-Series Data Using Wavelets. In *ICDE*. 212–221.
[17] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. 2012. Searching and mining trillions of time series subsequences under dynamic time warping. In *SIGKDD*. 262–270.
[18] Jin Shieh and Eamonn J. Keogh. 2008. *i*SAX: indexing and mining terabyte sized time series. In *SIGKDD*. 623–631.
[19] Jiaye Wu, Peng Wang, Ningting Pan, Chen Wang, Wei Wang, and Jianmin Wang. 2019. KV-Match: A Subsequence Matching Approach Supporting Normalization and Time Warping. In *ICDE*. 866–877.
[20] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2020. Massively Distributed Time Series Indexing and Querying. *IEEE Trans. Knowl. Data Eng.* 32, 1 (2020), 108–120.
[21] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. 2016. Matrix profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In *ICDM*.
[22] Byoung-Kee Yi and Christos Faloutsos. 2000. Fast Time Sequence Indexing for Arbitrary Lp Norms. In *VLDB*. 385–394.
[23] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for interactive exploration of big data series. In *SIGMOD*. 1555–1566.

Short Paper

Progressive Mergesort: Merging Batches of Appends into Progressive Indexes

Pedro Holanda
CWI, Amsterdam
holanda@cwi.nl

Stefan Manegold
CWI, Amsterdam
manegold@cwi.nl

ABSTRACT

Interactive exploratory data analysis consists of workloads that are composed of filter-aggregate queries with highly selective filters [1]. Hence, their performance is dependent on how much data they can skip during their scans, with indexes being the most efficient technique for aggressive data-skipping. Progressive Indexes are the state-of-the-art on automatic index creation for interactive exploratory data analysis. These indexes are partially constructed during query execution, eventually refining to a full index. However, progressive indexes have been designed for static databases, while in exploratory data analysis updates — usually batch-appends of newly acquired data — are frequent.

In this paper, we propose *Progressive Mergesort*, a novel merging technique to make Progressive Indexes cope with updates. Progressive Mergesort differs from other merging techniques for partial indexes as it incorporates the index budget strategy design from Progressive Indexing. It follows the same three principles as Progressive Indexes: (1) fast query execution, (2) high robustness,(3) guaranteed convergence.

Our experimental evaluation demonstrates that Progressive Mergesort is capable of achieving a 2x speedup when merging updates and up to 3 orders of magnitude lower variance than the state of the art.

1 INTRODUCTION

Data scientists perform interactive exploratory data analysis to discover unexpected patterns in large collections of data. This process is done using hypothesis-driven trial-and-error queries [10]. Given the result of a query, the data scientists refine their original hypothesis and either zoom in on the same data segment or move to a different one depending on the insights gained.

In the typical interactive exploratory data analysis workload, the data scientist inspects a massive amount of data by issuing selective analytical queries (usually via a visualization tool) to test their hypothesis. Battle et al. [1] depict that the most demanding type of interactive queries are cross filter applications (i.e., grouping data after applying selective filters). In these workloads, users expect almost immediate responses from the system, and each movement on the visualization tool will immediately submit another query to the database system.

Figure 1 depicts an example of a cross filter application. Here the data scientist uses a dataset that contains multiple attributes of flight information. The user visualizes each attribute as one histogram figure (e.g., departure time or airtime in minutes). The range slider on the top of the figure allows the users to change the filter used to construct these histograms, and the graphs are automatically updated depending on the new filter input.

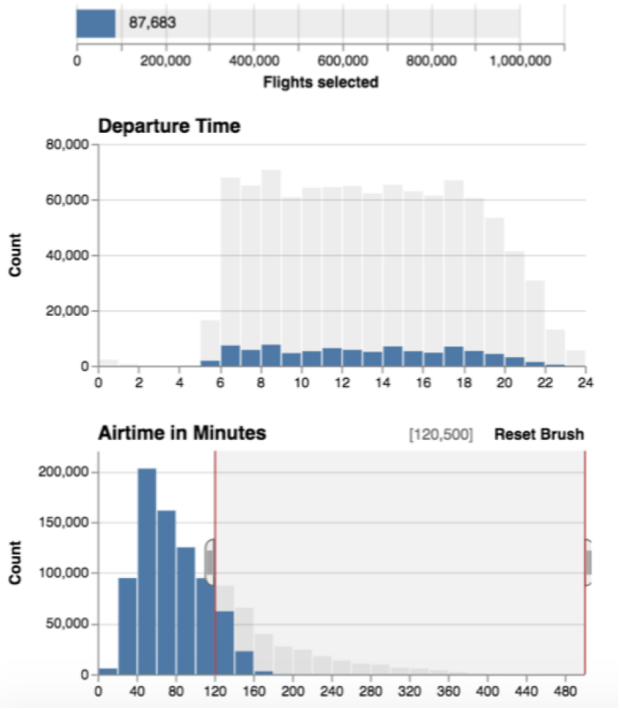


Figure 1: Interactive Data Analysis Example [1]

Since these workloads are dependent on a filter, when these filters are selective (e.g., wanting to know the information of a small number of flights), aggressive data skipping techniques can significantly improve the query performance.

Index structures are frequently used to boost workloads that depend on data skipping. There are two main strategies that automatically create indexes for interactive data analysis.

(1) Adaptive Indexing [6, 9] automatically creates indexes based on query predicates of range queries. They perform quicksort iterations with query predicates as pivots, indexing the accessed pieces during query execution, efficiently smearing out the index creation cost over a workload.

Adaptive Indexing follows a philosophy of only indexing the minimum amount of data necessary to the currently executing query. Although this strategy allows for fast convergence on skewed workloads (i.e., workloads where the same piece is frequently accessed), it has no control over the amount of indexing that one query can perform. When accessing pieces with different levels of refinement, query execution time spikes, resulting in a highly unpredictable query cost, which is undesirable for interactive data analysis since the user expects the query to be executed within a time limit.

(2) Progressive Indexes [3, 5, 8] are designed to be highly robust, have a predictable convergence, and present a low total cost during the entire workload execution. Their main difference

from Adaptive Indexing is the introduction of an indexing budget δ. With this indexing budget, the data scientist sets a value for δ, and the index invests a fixed amount of time into per-query index creation, being the state-of-the-art algorithm developed for interactive exploratory data analysis.

The major drawback of Progressive Indexes is that they are only designed for static-databases. However, in the interactive data analysis scenario, the data is not static but rather frequently updated with batches of data that must be appended. As an example, in our flight dataset we can consider the scenario where batches of data are regularly appended since new flights happen all the time (e.g., either data is appended every few minutes, hours, days, depending on how critical is to analyze recent data).

One way of adapting the current Progressive Indexing strategy to support updates is to use the techniques developed for merging updates on Adaptive Indexes since they produce similar partial-indexes up-to full index convergence. However, these merging techniques follow Adaptive Indexing's philosophy of lazy query execution, drastically decreasing robustness (i.e., it creates performance spikes that vary the per-query response time in orders of magnitudes up and down), with no guaranteed convergence and high penalties for larger batches of appends.

In this paper, we introduce *Progressive Mergesort*. Progressive Mergesort is designed to efficiently merge batches of appends while following the core design decisions of progressive indexing. It presents a low-query impact even for large batches, high robustness, and guaranteed convergence (i.e., all elements are merged into one array).

2 RELATED WORK

In this section, we will cover how Progressive Indexing, in particular progressive quicksort, works and will present the Adaptive Merges algorithms that merge updates into Adaptive Indexing.

2.1 Progressive Indexing

Progressive Indexes are inspired by Adaptive Indexes. Both techniques perform index creation during query execution, aiming to smear out the index creation cost over the workload. Consequently, the indexes produced by both techniques have a similar format (i.e., both are partial indexes), except that once fully converged, progressive indexing turns into a standard B-tree. One major difference between Progressive Indexes and Adaptive Indexes is that Progressive Indexes use an index budget constraint δ that indicates the amount of data that can be indexed, in one sorting iteration, per query, while Adaptive Indexes only performs full sorting iterations (e.g., adaptive indexing will fully partition one column around a pivot in one query, while progressive indexing will partition a δ fraction of the column.).

Progressive Indexes come in two flavors, the fixed-δ where the user picks a fraction of the data, and the same fraction is indexed per query, and a greedy version, where the user sets a desired query execution time. The greedy algorithm uses a cost-model to select a suitable δ tailored for each query automatically. In this paper, we will focus on the fixed-δ version of Progressive Indexing and will leave the greedy version as future work.

Figure 2 depicts an example of Progressive Quicksort with $\delta = 0.5$ (i.e., half the data is pivoted in each query), a progressive indexing technique inspired by the quicksort algorithm. Progressive quicksort is triggered when a filter is executed on a column. In its first phase (*Initialize*), an uninitialized array is allocated with the same space as the original column. A pivot is then selected, and the data is copied to either the bottom or the top of the new array considering the pivot. The subsequent step (*Initialize 2*) can already take advantage of this information and only scan the necessary parts (either top, bottom, neither, or both) relevant to the query. It also continues the copy process until our progressive indexing array is completely populated. At the end of the *initialize phase*, the column is partitioned into two pieces. In the example ≤ 10 and > 10. We now start the *refinement phase* where pivots are selected for each piece, and they are ordered in-place. When pieces are sufficiently small, we fully sort them. This phase results in a completely sorted array. When reaching a completely sorted array, the *consolidation phase* starts building a bottom-up B-Tree on top of the array.

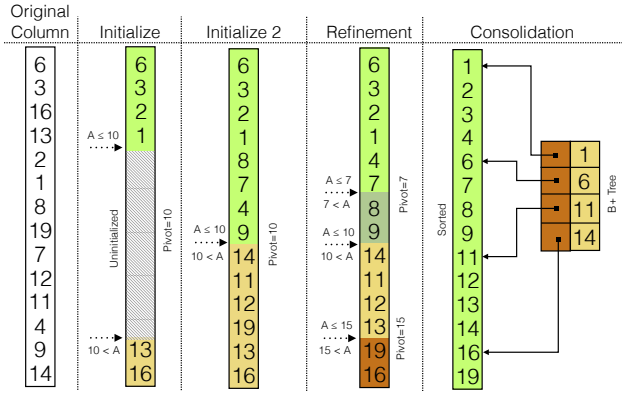


Figure 2: Progressive Quicksort [5]

2.2 Adaptive Merges

There are three main algorithms designed to efficiently merge appends into adaptive indexes [7], the *Merge Complete*, *Merge Gradual*, and *Merge Ripple*, and we will refer to these algorithms as *Adaptive Merges* from now on. They follow the same philosophy of Adaptive Indexing by only merging appends when necessary. They differ from each other in terms of what data they will merge and how they merge it. In the following subsections, we overview each algorithm and present an example of their execution. Besides the strategies to efficiently merge appends into the index's column, Holanda et al. [4] present a strategy to prune cold data from the cracker index to boost updates. However, we do not explore this strategy in this paper since it directly goes against our full convergence philosophy.

Merge Complete (MC) This algorithm completely merges the full *Appends* vector into the *Cracker Column* (i.e., the cracker column is a full copy of the original column owned by the adaptive indexing structure) as soon as a query requests data that is also present in the *Appends* vector.

Merge Gradual (MG) Merge Gradual differs from Merge Complete with respect to the amount of data merged per query. It only merges items that qualify for the currently executing query.

Merge Ripple (MR) Like Merge Complete, the Merge Ripple algorithm only merges the elements that qualify for the query predicates. They differ on how they merge them. In the Merge Ripple, instead of resizing the Cracker Column and appending the element to its end as its first step, it starts by swapping the to-be inserted element with the first element in the next greater-neighboring piece from its correct piece.

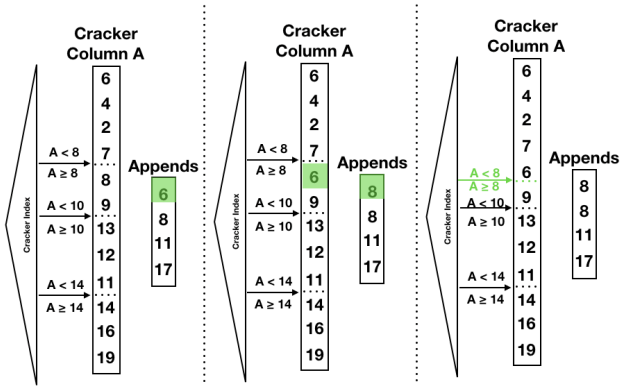


Figure 3: Merge Ripple on query A < 8

Figure 3 depicts an example of Merge Ripple executing the query $A < 8$. In our example, the column is already partitioned around three pivot points (8, 10, 14) and the appends array contains four values (6, 8, 11, 17). Since we only need to insert element 6 from the appends array, we perform a cracker index lookup and identify the element's piece (i.e., the first piece holding 6, 4, 2, and 7). We then go to the successor piece (i.e., piece 2 with elements 8 and 9) and swap the first element of that piece (8) with the element in our appends (6). After that, we only need to update the cracker index node that points to the value 8. In this case, we only had to perform 1 swap and update 1 node in the cracker index to merge 25% of our appends. Merge Ripple performs fewer swaps and updates than the previous algorithms while merging the necessary amount of data to our index.

3 PROGRESSIVE MERGESORT

Progressive Mergesort is a progressive indexing technique inspired by the mergesort algorithm [2] and used for merging appends into the main progressive indexing structure. It follows the three pillars of progressive indexes: (1) low impact on query execution, (2) robust performance, and (3) guaranteed convergence. It relies on an index-budget δ that represents the percentage of the data that is indexed per-query, guaranteeing that the same amount of effort will be distributed for the entire workload.

In practice, during query execution, the δ defined for our Progressive Indexing algorithm is used for both the main index structure and progressive mergesort.

Progressive mergesort follows two distinct canonical phases, the refinement phase, and the merge phase, which are described in this section.

Refinement. In the refinement phase, we can use any of the other proposed progressive indexing algorithms, getting the most performance depending on data distribution and workload. Our budget is used as described in the original Progressive Indexing paper [5] depending on the algorithm executing the refinement. In this paper, we decided to experiment with Progressive Quicksort as our algorithm of choice. Utilizing the other algorithms is left as an engineering exercise for future work.

Merge. At the end of the refinement phase of any progressive indexing algorithm, the result is a sorted list. When all merge chunks are fully sorted, we progressively merge them into one sorted chunk. We perform a progressive two-way-merge in order to merge said chunks.

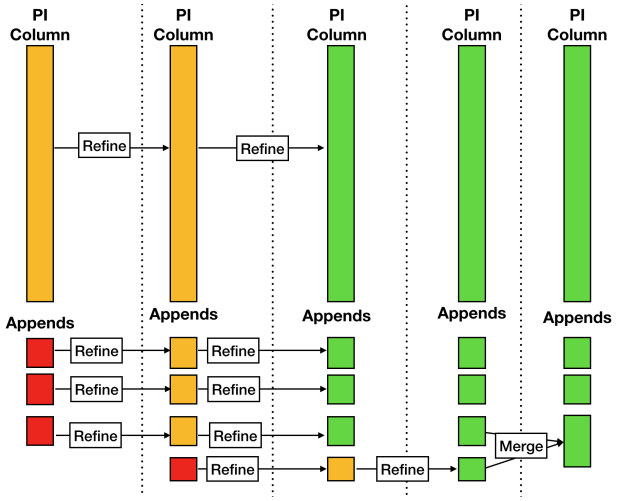


Figure 4: Progressive Mergesort

Figure 4 depicts a high-level concept of Progressive Mergesort. In this figure, red vectors are completely unsorted vectors, yellow are partially sorted vectors, and green are completely sorted. We start with our main index structure only partially sorted and with a new batch of appends.

It starts with the refinement phase. At this step, any Progressive Indexing technique can be used and will continue their execution until reaching completely sorted lists. When all chunks are entirely sorted, the second phase of Progressive Mergesort starts. Here, the *Appends* arrays are progressively merged into one array. One might note that new batches can be introduced while other batches are already being refined. In this case, a Progressive Mergesort run will be initiated to newly appended chunks. All these chunks use the same δ as our main progressive index but normalized to the chunk size. Only when the original Progressive Indexing column and the appends are fully sorted (i.e., we have one sorted column for the Progressive Indexing and one sorted column for all the appends) and the appends have the same or bigger size as the Progressive Indexing column we merge them.

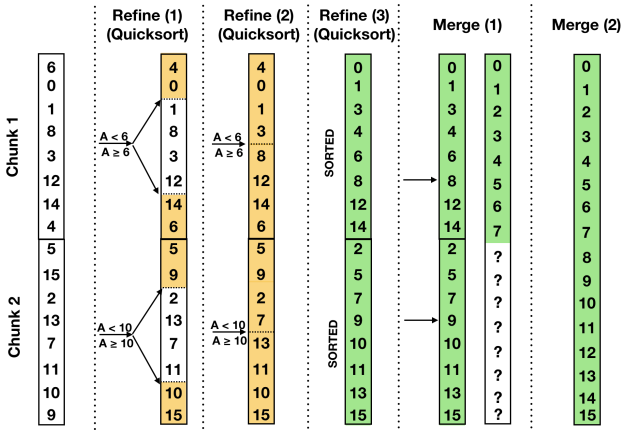


Figure 5: Progressive Mergesort Example ($\delta = 0.5$)

Figure 5 depicts an example of Progressive Mergesort with *delta* = 0.5. We start with two batches of updates. In the initial iterations, we execute Progressive Quicksort as the refinement phase. In *Refine (1)*, a Progressive Quicksort iteration is initiated

for each chunk, since $\delta = 0.5$ both iterations index half of each chunk around one pivot. In *Refine (3)* both Progressive Quicksort iterations ended, and both chunks are fully sorted. Hence we will start the merge phase of Progressive Mergesort in the following query. In *Merge (1)* we start to merge both lists using a two-way merge algorithm, and we stop when the resulting list is half complete due to our δ. For the chunks that are being merged, we must store the offsets where we stopped merging. Finally, in *Merge (2)* we end the merge phase with one completely sorted append list and delete the previous chunks.

Query Processing. When executing a query on a column with progressive indexing, we might encounter several arrays (i.e., the original Progressive Indexing column and batches of appends that started to be refined but are not yet merged) with different levels of refinement.

During the query execution, each array must be checked to return the elements that fit the query predicates. If the array is already fully sorted, a binary search will be executed to return the result. Otherwise, the array will be at some step of the refinement phase. Hence a lookup on the binary tree is necessary to return the offsets that match the query predicates.

When to Merge. In this paper, we decided to first completely merge all appends into one, fully sorted, append array. If this array has a size equal to or bigger than the current Progressive Indexing column, we merge both. This decision was made to avoid frequent resizes of large arrays (e.g., if we merged the Progressive Indexing column with every append first, this would result in a resize for the progressive column at every batch, which would be prohibitively expensive).

However, this decision is not necessarily optimal for all workloads. Having multiple arrays increase the random access to respond to the workload while diminishing the merge costs creating a trade-off depending on when and how these merges are performed. Creating an algorithm that decides when is the appropriate moment to merge these different arrays and which arrays should be merged is out of this paper's scope, and we leave it as future work.

4 EXPERIMENTAL ANALYSIS

This section provides an experimental evaluation of Progressive Mergesort and compares it with the Adaptive Merges techniques.

4.1 Setup

We implemented the Progressive Mergesort algorithm and the Adaptive Merges in a stand-alone program written in C++. The Progressive Mergesort uses Progressive Quicksort in its refinement phase.

Compilation. This application was compiled with GNU g++ version 7.2.1 using optimization level `-O3`.

Machine. All experiments were conducted on a machine equipped with 256 GB main memory and an 8-core Intel Xeon E5-2650 v2 CPU @ 2.6 GHz with 20480 KB L3 cache.

Appends. All experiments have 3 parameters regarding the appends, (1) the *batch_size* that represents the size of a batch of appends, (2) the *frequency* which represents an interval of queries where a new batch of appends is executed, and (3) *start_after* that describes how many queries need to be executed before the first append happens. With these 3 parameters we calculate the number of appends that will be executed $total\_appends = \frac{total\_queries - start\_after}{frequency} * batch\_size$, and divide our data set into the *original_column* set that represents our initially loaded column and the *appends* set that represent the appends that will be inserted.

Data set. We generate a synthetic data set composed of $N + total\_appends$ unique 8-byte integers, with $N \in \{10^7, 10^8, 10^9\}$ and representing the original column size. After generating the data set, we shuffle it following a uniform-random distribution and divide it into our original column and a list of appends.

Workload. Unless stated otherwise, all experiments consist of a synthetic workload with 10^4 queries in the form `SELECT SUM(R.A) FROM R WHERE R.A BETWEEN` V_1 `AND` V_2. A random value is selected for V_1 and $V_2 = V_1 + (N + total\_appends) * 1\%$.

Configuration. We experiment with 3 main configurations.

- High Frequency Low Volume (HFLV): A batch of appends with $batch\_size = 0.001\% * N$ executed every 10 queries.
- Medium Frequency Medium Volume (MFMV): A batch of appends with $batch\_size = 0.01\% * N$ executed every 100 queries.
- Low Frequency High Volume (LFHV): A batch of appends with $batch\_size = 0.1\% * N$ executed every 1000 queries.

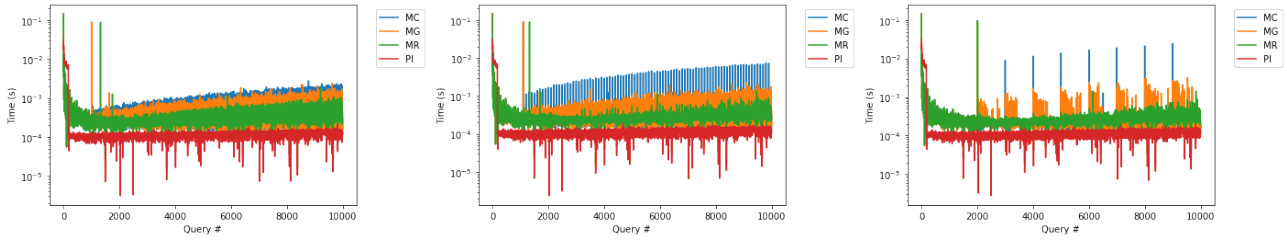
4.2 Performance Comparison

In this paper, we decided to use the Adaptive Merges algorithms only with Adaptive Indexing due to the increased complexity of implementing them to work with Progressive Indexing and leave this task as an engineering exercise for future work. Since the base indexing algorithm is different for the Adaptive Merges and Progressive Mergesort, we decided to start appending data after 1000 queries to have refined indexes and better isolate the actual append cost from early index creation. Hence we avoid the noise of partitioning the *original_column* and focus on the actual merges from the appends. Our Progressive Mergesort uses a fixed δ of 0.1 in all experiments. It is also important to notice that our Progressive Indexing implementation stops its convergence when it becomes a fully sorted list. We leave merging appends into the concise B+-Tree format as future work.

Figure 6 depicts a per-query performance comparison of Progressive Mergesort and Adaptive Merges. In this experiment, we use a data set with $N = 10^7$ and run all 3 configurations described in the previous section. We continue this section by describing two observations present in all experiments, (1) regarding the column resizes and (2) an overall analysis of query robustness.

Resizes. In all three configurations, HFLV, MFMV, and LFHV, we can notice that all three Adaptive Merges present a performance spike right after the start of the updates around query 1000. The main reason for this spike is the need to resize the *Cracker Column* when appending new data. Since this resize reserves 2 times the space of the original *Cracker Column*, it only happens once. It is also possible to notice that with Merge Ripple, the spike occurs 100 queries later than with Merge Complete and Merge Gradual. This is because Merge Ripple avoids resizing the *Cracker Column* by swapping the data from the *Appends* and the Column with the actual resize only happening when we are in the last piece. This problem does not exist with Progressive Mergesort since we perform a *vector.reserve()* to allocate memory to the merge vector, and filling out the merge vector is completed over multiple queries.

Robustness. The Merge Complete presents the lowest robustness from all algorithms. Whenever a merge happens, it has a big spike upwards since it completely merges it. Merge Gradual is the second-worst. Since it completely merges all elements that qualify the predicate, it does not have one big performance spike,



(a) HFLV (b) MFMV (c) LFHV

Figure 6: Progressive Mergesort and Adaptive Merges ($N = 10^7$ and $start\_after = 1000$)

spreading those merges through many queries. This is particularly visible in Figure 6c that depicts the low-frequency high volume experiment (i.e., at every 1000 queries, a batch of size 10^4 is inserted. One can see that at every 1000 queries, there is an upwards spike that slowly decreases for 500 queries, and then has a slop down since most of the *Appends* array was merged by that point. From the Adaptive Merges, the Merge Ripple presents the least variance. All queries slightly increase their cost with increasing updates. Finally, the Progressive Mergesort presents the lowest variance, with no performance spikes up.

One can notice that all algorithms present downwards spikes at the same queries overall 3 configurations. These are caused by noise due to the way we select our query predicates to fix our workload selectivity. Since we create our second query predicate as $V_2 = V_1 + (N + total\_appends) * 1\%$. Queries might not have exactly 1% selectivity if the data is not completely merged in the column. Since the figures are with the y-axis in log scale, small differences in the selectivity produce these downwards performance spikes.

4.3 Varying Data Sizes

	Workload	MC	MG	MR	PM
10^7	HFLV	2.72	3.52	2.57	1.07
	MFMV	2.18	3.39	2.45	1.07
	LFHV	2.00	2.55	2.34	1.06
10^8	HFLV	22.76	26.16	26.61	10.64
	MFMV	20.25	26.14	25.19	10.72
	LFHV	22.14	22.42	23.89	10.63
10^9	HFLV	209.25	221.67	295.39	104.77
	MFMV	206.39	219.39	267.94	104.96
	LFHV	197.89	200.62	250.62	103.95

Table 1: Cumulative Time (s)

Table 1 depicts the total execution cost for the workload, excluding the initial 1000 queries. On all experiments, Progressive Mergesort presented approximately 2x better performance than the best performing Adaptive Merge algorithm. The main reason for this performance difference is that all Merge Adaptive algorithms must keep the appends sorted to merge them efficiently. This problem impacts Merge Ripple the most since it tends to keep a larger appends array due to its lazier merging property. That means that a larger array must be re-sorted at every append insertion. One might notice that the results of Adaptive Merges seem to directly contradict Idreos et al. [7], where Merge Ripple was the best performing algorithm of the three. The HFLV with $N = 10^7$ is the only experiment with the same parameters as the original paper and showcases a similar result, with Merge Ripple being the fastest of the Adaptive Merges. However, as discussed before, with larger appends Merge Ripple starts to lose its benefit of fewer swaps to keep the append vector sorted.

One other interesting result is the variance in the total cost depending on the configuration of the workload. The Adaptive Merges algorithms present a much higher variance than Progressive Mergesort for the same data size. This is more prominent with larger data sizes. Taking $N = 10^9$ as an example, Merge Complete presents a variance of 11.36s, Merge Gradual of 21.05s, Merge Ripple of 44.72s, and Progressive Mergesort of 1.01s.

Compared to the Adaptive Merges algorithms, Progressive Mergesort has a very low variance from configurations at the same data size. This is due to the Progressive Mergesort algorithm not performing a complete sort in the append list but rather properly refining and merging it depending on their data size.

	Workload	MC	MG	MR	PM
10^7	HFLV	e-07	e-07	e-07	e-10
	MFMV	e-06	e-07	e-07	e-10
	LFHV	e-06	e-07	e-07	e-10
10^8	HFLV	e-05	e-05	e-05	e-07
	MFMV	e-05	e-05	e-05	e-07
	LFHV	e-04	e-05	e-05	e-07
10^9	HFLV	e-03	e-03	e-03	e-06
	MFMV	e-03	e-03	e-03	e-06
	LFHV	e-02	e-03	e-03	e-06

Table 2: Robustness (Orders of Magnitude)

Table 2 depicts the order of magnitude of the query variance of each workload on all 3 data sizes. We only calculate the query variance after executing the first 1000 queries. Note that the lower the variance the more robust the algorithm is. As expected, Merge Complete presents the lowest robustness since it completely merges the *Appends* array to the *Cracker Column* causing a huge performance spike. The Merge Gradual/Ripple are better than the Merge Complete, since it only merges that tuples that qualify the query predicates. Progressive Mergesort present the highest robustness due to its indexing budget, effectively offering a more fine-grained control over the stream of queries.

4.4 Appends during Index Creation

To perform a fair comparison of the Adaptive Merges and Progressive Mergesort, we only initiated the updates after 1000 queries to minimize the initial index creation cost of Adaptive Indexing and Progressive Indexing. However, after 1000 queries, the progressive indexing is already fully converged (i.e., the main index is a sorted list).

In this experiment, we want to evaluate Progressive Mergesort's impact during Progressive Indexing's creation phase (i.e.,

Initialization and Refinement phases). In our setup, we use a dataset with $N = 10^7$, a workload with 1% selectivity and 200 queries, and three different update setups. All update setups start at the first query and perform appends at every 10 queries, they differ on the size of the batches, with batches of size 100, 1000 and 10000.

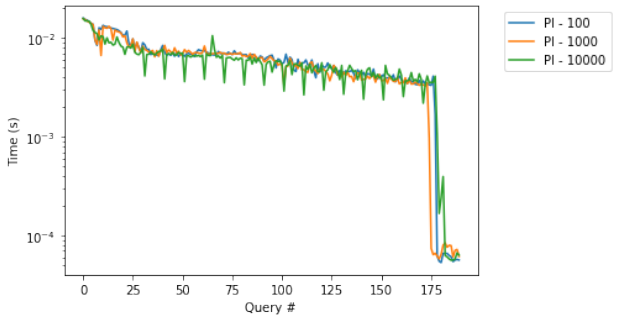


Figure 7: Progressive Mergesort before index convergence.

Figure 7 depicts the per-query cost for the 200 queries. The height of the performance spikes are strongly correlated to the batch sizes, with larger batches introducing a higher spike. This happens due to our strategy using a fixed δ (i.e., a % of the total size of the data that is indexed per-query) for the entire workload. Hence the more data we ingest, the actual per-query cost will increase since the data size increases. One way of minimizing this issue is to extend the cost models proposed in Holanda et al. [5] to automatically generate a value for δ to reduce query variance. We leave that algorithm as an exercise for future work.

5 CONCLUSION & FUTURE WORK

In this paper, we introduce the *Progressive Mergesort*, a novel progressive algorithm used to merge batches of appends. We compare it to the state-of-the-art merging algorithms from adaptive indexing techniques and show how they perform under multiple synthetic benchmarks. Our solution is more robust and faster than the state-of-the-art.

We point out the following as the main aspects to be explored in Progressive Mergesort's future work:

- **Integrating Merge Ripple With Progressive Indexing.** In our experiments we compare against adaptive indexing using the merge gradual/complete/ripple algorithms. However, this comparison would be even more significant if these algorithms were implemented directly into progressive indexing. For example, if the main index algorithm is Progressive Quicksort, by using an AVL-Tree, similar merge algorithms could be used.
- **Refinement Method.** In this paper, we only use Progressive Quicksort as our refinement strategy within Progressive Mergesort. However, in the Progressive Indexing work, it is demonstrated that different progressive indexing algorithms can present better performance depending on the data distribution and workload. With mergesort, we also have the opportunity of selecting a different algorithm for each chunk in the refinement step. Deciding which algorithm to use could drastically improve performance.
- **Merge Strategy.** Deciding when to merge and which arrays to merge can be beneficial to the cumulative cost of the workload since there is a trade-off on the random access vs merging costs (i.e., keeping many smaller arrays or frequently merging them in order to maintain only a small number of bigger arrays). Designing an algorithm that takes that this trade-off into consideration is left as future work.
- **Greedy Progressive Mergesort.** Our current implementation of progressive mergesort relies on a fixed δ for the entire workload. The development of a cost-model to the merge phase will allow it to also be integrated with progressive indexing algorithms that use an interactivity threshold and automatically adapt the δ value to boost robustness. Hence, as future work, a greedy version of our progressive mergesort can bring even fewer performance spikes to our algorithm.
- **Handling Updates.** In this paper, we describe how to efficiently merge appends, since these are the most common types of updates in interactive data analysis. However, although deletes and updates are not frequent, they might still occur, therefore progressive mergesort must be capable of properly handling them.
- **Multidimensional Updates.** Until now, we only focused on unidimensional progressive indexing. However, multidimensional progressive indexing [8] was recently proposed to efficiently index columns for queries with multiple selective filters. In this algorithm, a KD-Tree is used to store and navigate the partitions created by progressive indexing. To support updates on this structure, progressive mergesort must be extended to consider the KD-Tree nodes to merge multiple batches of updates correctly.
- **Real Benchmarks.** The Sloan Digital Sky Survey [1] is an open-source project that maps the universe with an open data set and interactive-exploratory query logs. Capturing the updates on this database can depict a good representation of real patterns of updates on interactive data.

ACKNOWLEDGMENTS

This work was funded by the Netherlands Organisation for Scientific Research (NWO), project "Data Mining on High-Volume Simulation Output (DAMIOSO)".

REFERENCES

[1] Leilani Battle, Philipp Eichmann, Marco Angelini, Tiziana Catarci, Giuseppe Santucci, Yukun Zheng, Carsten Binnig, Jean-Daniel Fekete, and Dominik Moritz. 2020. Database Benchmarking for Supporting Real-Time Interactive Querying of Large Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1571–1587.
[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.
[3] Pedro Holanda. 2018. Progressive Indices: Indexing Without Prejudice.. In *PhD@ VLDB*.
[4] Pedro Holanda and Eduardo Cunha de Almeida. 2017. SPST-Index: A Self-Pruning Splay Tree Index for Caching Database Cracking. In *EDBT*. 458–461.
[5] Pedro Holanda, Mark Raasveldt, Stefan Manegold, and Hannes Mühleisen. 2019. Progressive indexes: indexing for interactive data analysis. *PVLDB* 12, 13 (2019), 2366–2378.
[6] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*. 68–78.
[7] Stratos Idreos, Martin L Kersten, and Stefan Manegold. 2007. Updating a cracked database. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 413–424.
[8] Matheus Nerone, Pedro Holanda, Eduardo Almeida, and Stefan Manegold. 2021. Multidimensional Adaptive & Progressive Indexes. In *ICDE*.
[9] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Uncracked Pieces in Database Cracking. *PVLDB* 7, 2 (2013), 97–108. https://doi.org/10.14778/2732228.2732229
[10] Thibault Sellam, Emmanuel Müller, and Martin Kersten. 2015. Semi-Automated Exploration of Data Warehouses. In *CIKM*. 1321–1330.

[1] https://www.sdss.org/

Short Paper

Multiple-Source Context-Free Path Querying in Terms of Linear Algebra

Arseniy Terekhov
simpletondl@yandex.ru
Information Technologies,
Mechanics and Optics University
JetBrains Research
St. Petersburg, Russia

Vlada Pogozhelskaya
pogozhelskaya@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Vadim Abzalov
vadim.i.abzalov@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Timur Zinnatulin
teemychteemych@gmail.com
Saint Petersburg State University
St. Petersburg, Russia

Semyon Grigorev
s.v.grigoriev@spbu.ru
semyon.grigorev@jetbrains.com
Saint Petersburg State University
JetBrains Research
St. Petersburg, Russia

ABSTRACT

Context-Free Path Querying (CFPQ) allows one to express path constraints in navigational graph queries as context-free grammars. Although there are many algorithms for CFPQ developed, no graph database provides full-stack support of CFPQ. The Azimov's CFPQ algorithm is applicable for real-world graph analyses, as shown by Arseniy Terekhov. In this work we provide a modification to Azimov's algorithm for multiple-source CFPQ which makes the algorithm more practical and eases the integration into RedisGraph graph database. We also implement a Cypher graph query language extension for context-free constraints. Thus we provide the first full-stack support of CFPQ for graph databases. Our evaluation shows that the provided solution is suitable for real-world graph analyses.

1 INTRODUCTION

Language-constrained path querying [2] is a way to search for paths in edge-labeled graphs where constraints are formulated in terms of a formal language. The language restricts the set of accepted paths: the sentence formed by the labels of a path should be in the language. Regular languages are the most popular class of constraints used as navigational queries in graph databases. In some cases, regular languages are not expressive enough and context-free languages are used instead. Context-free path querying (CFPQ), can be used for RDF analysis [23], biological data analysis [18], static code analysis [16, 24], and in other areas.

CFPQ have been studied a lot since the problem was first stated by Mihalis Yannakakis in 1990 [22]. Jelle Hellings investigates various aspects of CFPQ in [6–8]. A number of CFPQ algorithms were proposed: (G)LL and (G)LR-based algorithms by Ciro M. Medeiros et al. [12], Fred C. Santos et al. [17], Semyon Grigorev et al. [5], and Ekaterina Verbitskaia et al. [20]; CYK-based algorithm by Xiaowang Zhang et al. [23]; combinators-based approach to CFPQ by Ekaterina Verbitskaia et al. [21]. Nevertheless, the application of context-free constraints for real-world data analysis still faces many problems. The first problem is bad performance of the proposed algorithms on real-world data, as shown by Jochem Kuijpers et al. [11]. The second problem is that no graph database provides full-stack support of CFPQ, since most effort was made in developing algorithms and researching their theoretical properties. This fact hinders research of problems which can be reduced to CFPQ, thus it hinders the development of new solutions for them. For example, graph segmentation in data provenance analysis was recently reduced to CFPQ [14], but the evaluation of the proposed approach was complicated by the fact that no graph database supported CFPQ.

Rustam Azimov proposed a matrix-based algorithm for CFPQ in [1]. This algorithm provides a solution performant enough for real-world data analyses, as shown by Nikita Mishim et al. in [15] and Arseniy Terekhov et al. in [19]. This algorithm computes reachability and provides a single path which satisfies constraints for *every* vertex pair in the graph. Namely it solves *all-pairs* context-free path querying problem. In many real-world scenarios it is redundant to handle all possible pairs, instead one can provide one or a relatively small set of start vertices.

While all-pairs context-free path querying is a problem well studied, best to our knowledge, there is no solutions for the single-source and multiple-source CFPQ. In this work we propose a matrix-based *multiple-source* (and *single-source* as a partial case) CFPQ algorithm and provide full-stack support of CFPQ based on the proposed algorithm.

To sum up, we make the following contributions in this paper.

(1) We modify the Azimov's matrix-based CFPQ algorithm and provide a multiple-source matrix-based CFPQ algorithm. As a partial case, it is possible to use our algorithm in a single-source scenario. Our modification is still based on linear algebra, hence it is simple to implement and allows one to use high-performance libraries and utilize modern parallel hardware for queries evaluation.
(2) We provide full-stack support of CFPQ by extending the RedisGraph[1] [3] graph database. To do it, we implement a Cypher query language extension[2] that makes it possible to use context-free constraints, implemented the proposed algorithm in a RedisGraph backend, and supported the

[1] RedisGraph graph database Web-page: https://redislabs.com/redis-enterprise/redis-graph/. Access date: 19.07.2020.
[2] Proposal which describes path patterns specification syntax for Cypher query language: https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc. The proposed syntax allows one to specify context-free constraints. Access date: 19.07.2020.

new syntax in the RedisGraph query execution engine. As far as we know, it is the first full-stack implementation of CFPQ. Finally, we evaluate the proposed solution and show that it is performant and memory-efficient enough to be applicable for real-world graph querying.

2 PRELIMINARIES

In this section we introduce common definitions in graph theory and formal language theory which are used in this paper. Also, we provide a brief description of Azimov's algorithm which is used as a base of our solution.

2.1 Basic Definitions of Graph Theory

In this paper we use a labeled directed graph as a data model and define it as follows.

Definition 2.1. Labeled directed graph is a tuple of six elements $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$, where

- V is a finite set of vertices. For simplicity, we assume that the vertices are natural numbers ranging from 0 to $|V| - 1$.
- $E \subseteq V \times V$ is a set of edges.
- Σ_V and Σ_E are sets of labels of vertices and edges respectively, such that $\Sigma_V \cap \Sigma_E = \varnothing$.
- $\lambda_V : V \to 2^{\Sigma_V}$ is a function that maps a vertex to a set of its labels, which can be empty.
- $\lambda_E : E \to 2^{\Sigma_E} \setminus \{\varnothing\}$ is a function that maps an edge to a non-empty set of its labels, so each edge must have at least one label.

Labeled graph is the basis of the widely-used *property graph* data model and allows one to use not only edge labels but also vertex labels in navigation queries.

An example of the labeled directed graph D_1 is presented in figure 1. Here the sets of labels $\Sigma_V = \{x, y\}$ and $\Sigma_E = \{a, b, c, d\}$. We omit the sets of vertex labels whenever they are empty.

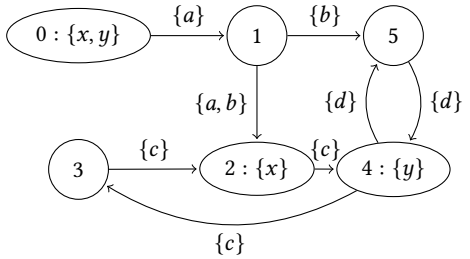


Figure 1: The input graph D_1

Definition 2.2. Path π in the graph $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$ is a finite sequence of vertices and edges $(v_0, e_0, v_1, e_1, ..., e_{n-1}, v_n)$, where $\forall i, 0 \le i \le n : v_i \in V; \forall j, 1 \le j \le n : e_j = (v_j, v_{j+1}) \in E$.

Definition 2.3. An *adjacency matrix* M of the graph D is a matrix of size $|V| \times |V|$, such that

$$M[i,j] = \begin{cases} \lambda_E((i,j)) & , (i,j) \in E \\ \varnothing & , otherwise \end{cases}$$

The adjacency matrix M of the graph D_1 (fig. 1) is the following:

$$M = \begin{pmatrix} \varnothing & \{a\} & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{a,b\} & \varnothing & \varnothing & \{b\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{c\} & \varnothing \\ \varnothing & \varnothing & \{c\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{c\} & \varnothing & \{d\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{d\} & \varnothing \end{pmatrix}.$$

Definition 2.4. Let M be an adjacency matrix of the graph D. Then the *adjacency matrix of label* $l \in \Sigma_E$ of graph D is a matrix $\mathcal{E}^l$ of size $|V| \times |V|$, such that

$$\mathcal{E}^l[i,j] = \begin{cases} 1 & , l \in M[i,j] \\ 0 & , otherwise \end{cases}$$

Definition 2.5. A *boolean decomposition of adjacency matrix* M of the graph D is a set of Boolean matrices $\mathcal{E} = \{\mathcal{E}^l \mid l \in \Sigma_E\}$, where $\mathcal{E}^l$ is the adjacency matrix of label l.

For example, the boolean decomposition of the adjacency matrix M of the graph D_1 is the set of matrices $\mathcal{E}^a, \mathcal{E}^b, \mathcal{E}^c, \mathcal{E}^d$:

$$\mathcal{E}^a = \begin{pmatrix} \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \ \mathcal{E}^b = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

$$\mathcal{E}^c = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \ \mathcal{E}^d = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Definition 2.6. A *vertex label matrix* H of the graph D is a matrix of size $|V| \times |V|$, such that

$$H[i,j] = \begin{cases} \lambda_V(i) & , i = j \\ \varnothing & , otherwize \end{cases}$$

The vertex label matrix H of the example graph D_1 is

$$H = \begin{pmatrix} \{x,y\} & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{x\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{y\} & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \end{pmatrix}.$$

Definition 2.7. Let H be a vertex label matrix of graph D. Then the *vertices matrix of label* l is a matrix $\mathcal{V}^l$ of size $|V| \times |V|$, such that

$$\mathcal{V}^l[i,j] = \begin{cases} 1 & , l \in H[i,j] \\ 0 & , otherwise \end{cases}$$

Definition 2.8. A *boolean decomposition of vertex label matrix* H of the graph D is the set of Boolean matrices $\mathcal{V} = \{V^l \mid l \in \Sigma\}$, where $\mathcal{V}^l$ is a vertices matrix of label l.

Vertex label matrix H of the graph D_1 can be decomposed into a set of the following Boolean matrices:

$$\mathcal{V}^x = \begin{pmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \ \mathcal{V}^y = \begin{pmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

2.2 Basic Definitions of Formal Languages

We use context-free grammars as paths constraints, thus in this subsection we define context-free languages and grammars.

Definition 2.9. A *context-free grammar* G is a tuple (N, Σ, P, S), where

- N is a finite set of nonterminals
- Σ is a finite set of terminals, $N \cap \Sigma = \varnothing$
- P is a finite set of productions of the form $A \to \alpha$, where $A \in N$, $\alpha \in (N \cup \Sigma)^*$
- S is the start nonterminal

Definition 2.10. A *context-free language* is a language generated by a context-free grammar G: $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\underset{G}{\Longrightarrow}} w\}$

Where $S \overset{*}{\underset{G}{\Longrightarrow}} w$ denotes that a string w can be generated from a starting non-terminal S using a sequence of productions from P.

Definition 2.11. A context-free grammar $G = (N, \Sigma, P, S)$ is in *weak Chomsky normal form* (WCNF) if every production in P has one of the following forms:

- $A \to BC$, where $A, B, C \in N$
- $A \to a$, where $A \in N, a \in \Sigma$
- $A \to \varepsilon$, where $A \in N$

Note that weak Chomsky normal form differs from Chomsky normal form in the following:

- ε can be derived from any non-terminal;
- S can occur in the right-hand side of productions.

The matrix-based CFPQ algorithms process grammars only in weak Chomsky normal form, but every context-free grammar can be transformed into the equivalent grammar in this form.

Consider the context-free grammar $G_1 = (\{S\}, \{c, d, y\}, P, S)$, where P contains two rules: $S \to c\ S\ d$; $\quad S \to c\ y\ d$.

This grammar generates the context-free language:

$$L(G_1) = \{c^n y d^n, n \in \mathbb{N}\}.$$

The following grammar G_1^{wcnf} is a result of the transformation of G_1 to weak Chomsky normal form:

$$S \to C\ E \qquad E \to Y\ D \qquad C \to c \qquad D \to d$$
$$S \to C\ S_1 \qquad S_1 \to S\ D \qquad Y \to y$$

2.3 Context-Free Path Querying

Definition 2.12. Let $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$ be a labeled graph, $G = (N, \Sigma_V \cup \Sigma_E, P, S)$ be a context free grammar. Then a *context free relation* with grammar G on the labeled graph D is the relation $R_{G,D} \subseteq V \times V$:

$$R_{G,D} = \{(v_1, v_n) \in V \times V \mid \exists \pi = (v_1, e_1, v_2, \ldots, e_n, v_n) \in \pi(D) : l(\pi) \cap L(G) \neq \varnothing\},$$

where $l(\pi) \subset (\Sigma_V \cup \Sigma_E)^*$ is the set of labels along the path π:

$$l(\pi) = \lambda_V(v_1)^* \cdot \lambda_E(e_1) \cdot \lambda_V(v_2)^* \cdot \ldots \cdot \lambda_E(e_n) \cdot \lambda_V(v_n)^*$$

For example, π is a path from vertex 2 to vertex 5 in the labeled graph presented in figure 1: $\pi = 2 : \{x\} \xrightarrow{\{c\}} 4 : \{y\} \xrightarrow{\{d\}} 5$.

Labels along π form the set of sequences $l(\pi) = \{x^m c y^n d \mid n \geq 0, m \geq 0\}$. Only one of these sequences satisfies context-free constraints of the grammar G_1: cyd. Hence $l(\pi) \cap L(G_1) \neq \varnothing$ and the pair $(3, 6) \in R_{G_1,D}$.

Note that the definition of path labels allows for zero or more repetitions of a label of each vertex. This makes it possible to omit vertex labels or, if there are many vertex labels, to use them in an arbitrary order. It also permits to write a query which uses one vertex label multiple times. This definition may appear strange in some cases, but it depends on the semantics of the graph query language. Semantics formalization is planned for a future work, so we will stick to this definition in this paper.

Finally, we can define context-free path querying problem.

Definition 2.13. Context-free path querying problem is the problem of finding context-free relation $R_{G,D}$ for a given directed labeled graph D and a context-free grammar G.

In other words, the result of context-free path query evaluation is a set of vertex pairs such that there is a path between them and this path forms a word from the given language.

The context-free relation R_{G_1,D_1} for the graph D_1 and the context-free free grammar G_1 is the following:

$$R_{G_1,D_1} = \{(2, 4), (2, 5), (3, 4), (3, 5), (4, 4), (4, 5)\}.$$

Note that any relation $R_{G,D}$ can be represented as a Boolean matrix: $T[i, j] = 1 \iff (i, j) \in R_{G,D}$. In our example, R_{G_1,D_1} can be represented as follows:

$$T = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Definition 2.14. Suppose Src is a given set of start vertices, then *multiple-source context-free path querying problem* for the given Src, directed labeled graph D and context-free grammar G is to find a context-free relation $R_{G,D}^{Src} \subseteq Src \times V \subseteq R_{G,D}$. Thus we restrict start vertices of the paths of interest to be vertices from the given set.

As a special case, a *single-source CFPQ* is when Src is a singleton set. If we set $Src = \{2\}$ in the previous example, then the result is: $R_{G_1,D_1}^{\{2\}} = \{(2, 4), (2, 5)\}$.

2.4 Matrix-Based Algorithm

Our algorithm is based on the Azimov's CFPQ algorithm [1] which is based on matrix operations. This algorithm reduce CFPQ to operations over Boolean matrices and as a result allows one to use high-performance linear algebra libraries and utilize modern parallel hardware for CFPQ. Moreover, utilization of Boolean matrices simplifies the implementation of the algorithm.

Note, that the algorithm computes not only the context-free relation $R_{G,D}$ but also a set of context-free relations $R_{A,D} \subseteq V \times V$ for every $A \in N$. Thus it provides information about paths which form words derivable from any nonterminal in the given grammar. Also, this algorithm handles only the edge labels.

As was shown by Nikita Mishin et al. [15] and Arseniy Terekhov et al. [19], this algorithm can be implemented using various high-performance programming techniques (including GPGPU utilization), and it is applicable for real-world graph analysis. But this algorithm solves *all-pairs* version of CFPQ: it finds all pairs of vertices in the given graph, such that there exist a path between them which forms a word in the given language. Thus it is impractical in cases when we are only interested in paths which start from the specific set of vertices, especially if this set is relatively small. Moreover, Azimov's algorithm operates over an adjacency matrix of the whole input graph, and as a result it requires a huge amount of memory which may be a problem for a real-world graph database.

3 MATRIX-BASED MULTIPLE-SOURCE CFPQ ALGORITHM

In this section we introduce a multiple-source matrix-based CFPQ algorithm. This algorithm is a modification of Azimov's matrix-based algorithm for CFPQ and its core idea is to cut off those vertices which are not in the selected set of start vertices.

Let $G = (N, \Sigma, P, S)$ be the input context-free grammar, $D = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$ be the input graph and Src be the input set of start vertices. The result of the algorithm is a Boolean matrix which represents relation $R_{G,D}^{Src}$.

Algorithm 1 Multiple-source CFPQ algorithm

```
1:  function MultiSrcCFPQNaive(
        D = (V, E, Σ_V, Σ_E, λ_V, λ_E),
        G = (N, Σ, P, S),                          ▷ Grammar in WCNF
        Src)
2:      T ← {T^A | A ∈ N, T^A[i, j] ← false, for all i, j}
3:      TSrc ← {TSrc^A | A ∈ N, TSrc^A[i, j] ← false, for all i, j}
4:      for all v ∈ Src do                         ▷ Input matrix initialization
5:          TSrc^S[v, v] ← true
6:      MSrc ← TSrc^S
7:      for all A → x ∈ P | x ∈ Σ_E do             ▷ Simple rules initialization
8:          for all (v, to) ∈ E | x ∈ λ_E(v, to) do
9:              T^A[v, to] ← true
10:     for all A → x ∈ P | x ∈ Σ_V do
11:         for all v ∈ V | x ∈ λ_V(v) do
12:             T^A[v, v] ← true
13:     while T or TSrc is changing do             ▷ Algorithm's body
14:         for all A → BC ∈ P do
15:             M ← TSrc^A * T^B
16:             T^A ← T^A + M * T^C
17:             TSrc^B ← TSrc^B + TSrc^A
18:             TSrc^C ← TSrc^C + getDst(M)
19:     return MSrc * T^S
20: function getDst(M)
21:     A[i, j] ← false
22:     for all (v, to) ∈ V^2 | M[v, to] = true do
23:         A[to, to] ← true
24:     return A
```

In order to solve the single-source and multiple-source CFPQ problem, we modified the Azimov's algorithm. Each time, when a grammar rule is applied, only vertices of interest should be stored. To do it, we added one more matrix multiplication: $T^A = T^A + (TSrc^A \cdot T^B) \cdot T^C$, where $TSrc^A$ is a matrix of start vertices for the current iteration (lines **15-16** of the Algorithm 1). In the end of each iteration of the for loop, it is necessary to update the set of source vertices. To do it, we call the function getDst (see lines **20-24**), in line **18**. Thus, the modified algorithm supports the frontier of the vertices of interest and updates it on each iteration. Thus, it only computes the paths which start from the small set of selected vertices.

4 CFPQ FULL-STACK SUPPORT

To provide full-stack support of CFPQ, it is necessary to choose an appropriate graph database. It was shown by Arseniy Terekhov et al. [19] that matrix-based algorithm can be naturally integrated into RedisGraph because the algorithm and the database both operate over a matrix representation of graphs. Moreover, RedisGraph supports Cypher as a query language and there is a proposal which describes Cypher extension for context-free constraints. Thus we chose RedisGraph as a base for our solution.

4.1 Cypher Extension

The first thing to do is to extend the Cypher parser to support the context-free constraints. Tobias Lindaaker proposed an extension for context free constraints to the Cypher syntax[3], which is not implemented in the Cypher parsers yet.

This extension introduces path patterns, which are a powerful alternative to the original Cypher relationship patterns. Path patterns allow one to express regular constrains over the basic patterns such as relationship and node patterns. Like relationship patterns, they can be specified in the `MATCH` clause.

Listing 2 Query based on the example grammar G_1 written in Cypher with path patterns

```
1: PATH PATTERN S = ()-/ [:c ~S :d] | [:c (:y) :d] /->()
2: MATCH (v:x)-[:a | :c]->()-/ :b ~S /->(to)
3: RETURN v, to
```

The feature which allows one to specify context-free constraints is *named path patterns*: a path pattern can be assigned a name which can be used in other patterns or within the same pattern. Named patterns is defined in the `PATH PATTERN` clause. Using this feature, the structure of queries is pretty similar to a grammar in the Extended Backus-Naur Form (EBNF) [9].

An example of a query which uses named path patters is presented in listing 2. This query is based on the context-free grammar G_1. Namely, the path pattern `S` specifies exactly the same constraint as the grammar G_1. The `MATCH` clause consists of the relation pattern `[:a | :c]` and the path pattern `/:b ~S/`, and this path pattern references the named pattern `S`. The constraint specifies that a path of interest starts in a vertex labelled `x`, goes through an edge labelled either `a` or `c`, then the rest of the path is constrainted by a path pattern which starts with an edge `b` and follows with a path matched with `S`. The `RETURN` clause specifies what the result of the query is supposed to be. For the example graph D_1, this query returns the set of vertex pairs $\{(0, 4), (0, 5)\}$.

RedisGraph database supports a subset of the Cypher language and uses `libcypher-parser`[4] library to parse queries. We extend this library with the new syntax described in the proposal. Note that we implement[5] the complete syntax extension, not only the part necessary for simple CFPQ.

4.2 RedisGraph Extension

We implemented the multiple-source algorithm in the RedisGraph. We partially supported the proposed syntax extension in RedisGraph query execution engine so that one can specify the labels of edges and vertices and use named path patterns.

Processing the input as a whole may require a lot of memory. RedisGraph implements lazy evaluation: it creates execution strategy in terms of elementary operations each of which processes the input sequentially in *chunks*. This reduces memory consumption so that it does not depend on the input size which is crucial when dealing with big real-world graphs. However processing chunks comes with a time overhead. By changing the size of a chunk, a developer may adjust the ratio between the time and memory consumption so that it fits their needs.

We use subsets of the start vertices as chunks since it is most natural in the multiple source algorithm. We study how the size of a chunk affects the performance in the evaluation.

4.3 Evaluation

For RedisGraph evaluation, we used a PC with Ubuntu 18.04 installed. It has Intel Core i7-6700 CPU, 3.4GHz, and DDR4 64Gb RAM. RedisGraph with our extensions is installed[6].

[3]Formal syntax specification: https://github.com/thobe/openCypher/blob/rpq/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc#11-syntax. Access date: 19.07.2020.

[4]The `libcypher-parser` is an open-source parser library for Cypher query language. GitHub repository of the project: https://github.com/cleishm/libcypher-parser. Access date: 19.07.2020.

[5]The modified libcypher-parser library with support of syntax for path patterns: https://github.com/YaccConstructor/libcypher-parser. Access date: 19.07.2020.

[6]Sources of RedisGraph database with full-stack CFPQ support:https://github.com/YaccConstructor/RedisGraph/tree/path_patterns_dev. Access date: 19.07.2020.

4.3.1 Data Preparation. We use the graphs and respective queries g_1 and *geo* from [19] to evaluate the RedisGraph-based solution. The graphs are loaded into the RedisGraph database so that each vertex has a unique property `id` in the range $[0, \ldots, |V|-1]$, where $|V|$ is a number of vertices in the graph to load. This allows us to generate queries for a start vertex set with specific size using templates. The template for the g_1 query is provided in listing 3. Here `{id_from}` and `{id_to}` are placeholders for the lower and the upper bounds for `id`.

Listing 3 Cypher query pattern for g_1

```
PATH PATTERN S =
        ()-/ [<:SubClassOf [~S | ()] :SubClassOf]
        | [<:Type [~S | ()] :Type] /->()
MATCH (src)-/ ~S /->()
WHERE {id_from} <= src.id and src.id <= {id_to}
RETURN count(*)
```

We implemented a query generator for the queries g_1 and *geo* to create concrete queries for all the start sets which are used in the previous experiment.

4.3.2 Evaluation Results. We use *geo* query for *geospecies* graph as one of the hardest queries, and g_1 query for other graphs. We measure time and memory consumption for each start set.

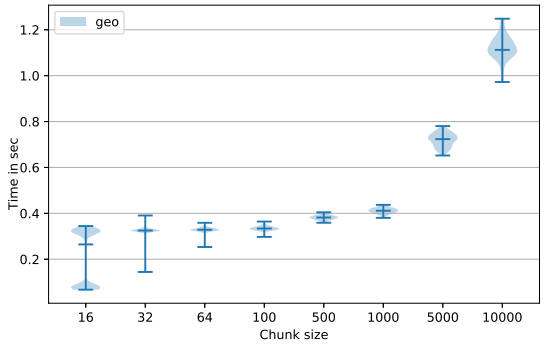


Figure 2: RedisGraph performance on *geospecies* graph

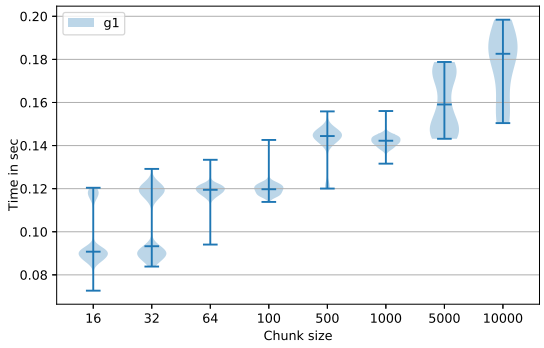


Figure 3: RedisGraph performance on *eclass_514en* graph

The execution time for all sets, except the set of size 10 000 for *geospecies* graph (fig. 2), is less than 1 second. Moreover, for smaller graph (*eclass_514en*), processing time is less than 0.2 second for all chunks (fig. 2).

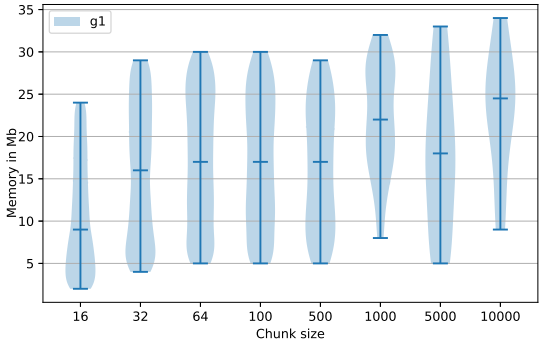


Figure 4: Memory consumption on *eclass_514en*

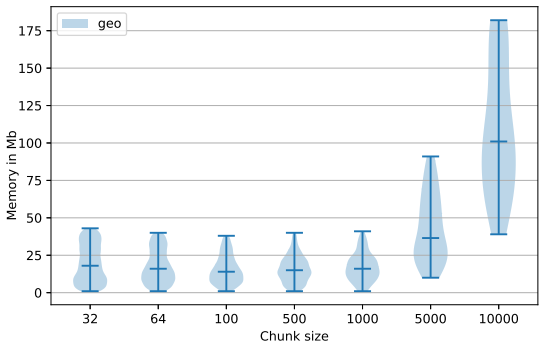


Figure 5: Memory consumption on *geospecies*

Memory consumption for the big graphs *eclass_514en* and *geospecies* is presented in figures 4 and 5 respectively. The amount of memory used depends on the graph and the query, but RedisGraph uses less that 50Mb of RAM to process graphs with relatively small chunks (≤ 1000). Note that RedisGraph includes memory management system, thus we measured all allocated memory, not only the memory really used for the query evaluation. As a result, we can conclude that the multiple-source CFPQ is significantly more memory efficient than creation of the complete reachability index and its filtering: processing the set of size 10 000 on *geospecies* graph requires less than 200Mb, while full index creation requires 16Gb [19].

We also evaluate how chunking affects the performance on the all-pairs reachability problem. We fix the size of a chunk to be 1000 for graphs of different sizes and measure time and memory required to process queries. Namely, we evaluate the query which is similar to the query from the previous scenario, but it does not constraint vertices ids (it does not have the `WHERE` clause). We measure total processing time (in seconds) and total required memory (in Mb). Also, we compare our solution with the results of Arseniy Terekhov et al. from [19] in which the Azimov's algorithm was naively integrated with RedisGraph without support of lazy query evaluation and query language. Similar hardware and the same input graphs and queries were used. Results are provided in table 1.

Although chunk-by-chunk processing is slower, it still requires reasonable time. Moreover, if the chunk size is comparable with the graph size (*core* and *pathways* graphs), then the execution time is comparable with the monolithic processing. Thus one can decrease execution time by increasing the chunk size. On the other hand, even with relatively small chunks (*eclass_514*, *go* and *geospecies* graphs), when for chunk-by-chunk processing is

Table 1: Full graph processing with chunks of size 1000

Graph	#V	#E	Q	Chunks		Mono [19]
				T (sec)	Mem (Mb)	T (sec)
core	1323	4342	g_1	0.003	2	0.004
pathways	6238	18 598	g_1	0.031	6	0.011
gohierarchy	45 007	980 218	g_1	0.847	62	0.091
enzyme	48 815	109 695	g_1	0.698	13	0.018
eclass_514en	239 111	523 727	g_1	18.825	35	0.067
geospecies	450 609	2 311 461	*geo*	80.979	196	7.146
go	272 770	534 311	g_1	72.034	40	0.604

100 times slower, our results are still reasonable for some cases. For example, it requires over 70 times less time for *geospecies* graph processing than the solution of Jochem Kuijpers et al. [11] which is based on Neo4j and requires more than 6000 seconds. Moreover, while the solution from [19] requires huge amount of memory (more than 16Gb for *geospecies* graph and *geo* query), our solution requires only 196Mb. We argue, that our solution is more suitable for general-purpose graph databases. First of all, the core scenario when the set of start vertices is relatively small can be handled efficiently. Second, all-pairs reachability, which is not a massive case, can be solved in reasonable time with low memory consumption. One can easily tune our solution to get the optimal time and memory consumption for their specific case.

5 CONCLUSION AND FUTURE WORK

In this paper we propose a multiple-source modifications of Azimov's CFPQ algorithm and utilize it to provide full-stack support of CFPQ. For our solution, we implement a Cypher extension as a part of `libcypher-parser`, integrate the proposed algorithm into RedisGraph, and extend RedisGraph execution plan builder to support the extended Cypher queries. We demonstrate that our solution is applicable for real-world graph analyses.

In the future, it is necessary to provide formal translation of Cypher to linear algebra, or to determine a maximal subset of Cypher which can be translated to linear algebra. There is a number of works on the translation of a subset of SPARQL to linear algebra, such as [4, 10, 13]. Most of them are practical-oriented and do not provide full theoretical basis to translate querying language to linear algebra. Others discuss only partial cases and should be extended to cover real-world query languages. Deep investigation of this topic can help to determine the restrictions of linear algebra utilization for graph databases.

ACKNOWLEDGEMENTS

The research was supported by the Russian Science Foundation, grant №18-11-00100.

We thank Roi Lipman for his help with investigation of the RedisGraph internals and pointing out the impractical memory consumption of the original Azimov's algorithm which gave us the motivation to develop the presented solution.

We thank Ekaterina Verbitskaia for the fruitful discussion and feedback which helped us to improve the paper.

REFERENCES

[1] Rustam Azimov and Semyon Grigorev. 2018. Context-free Path Querying by Matrix Multiplication. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '18)*. ACM, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3210259.3210264

[2] C. Barrett, R. Jacob, and M. Marathe. 2000. Formal-Language-Constrained Path Problems. *SIAM J. Comput.* 30, 3 (2000), 809–837. https://doi.org/10.1137/S0097539798337716 arXiv:https://doi.org/10.1137/S0097539798337716

[3] P. Cailliau, T. Davis, V. Gadepally, J. Kepner, R. Lipman, J. Lovitz, and K. Ouaknine. 2019. RedisGraph GraphBLAS Enabled Graph Database. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 285–286.

[4] Roberto De Virgilio. 2012. A Linear Algebra Technique for (de)Centralized Processing of SPARQL Queries. In *Conceptual Modeling*, Paolo Atzeni, David Cheung, and Sudha Ram (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 463–476.

[5] Semyon Grigorev and Anastasiya Ragozina. 2017. Context-free Path Querying with Structural Representation of Result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3166094.3166104

[6] Jelle Hellings. 2014. Conjunctive context-free path queries. In *Proceedings of ICDT'14*. 119–130.

[7] Jelle Hellings. 2015. Path Results for Context-free Grammar Queries on Graphs. *CoRR* abs/1502.02242 (2015). arXiv:1502.02242 http://arxiv.org/abs/1502.02242

[8] Jelle Hellings. 2015. Querying for Paths in Graphs using Context-Free Path Queries. *arXiv preprint arXiv:1502.02242* (2015).

[9] ISO/IEC. 1996. International Standard EBNF Syntax Notation. http://www.iso.ch/cate/d26153.html. 14977 edn. Online.

[10] Fuad Jamour, Ibrahim Abdelaziz, and Panos Kalnis. 2018. A Demonstration of MAGiQ: Matrix Algebra Approach for Solving RDF Graph Queries. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1978–1981. https://doi.org/10.14778/3229863.3236239

[11] Jochem Kuijpers, George Fletcher, Nikolay Yakovets, and Tobias Lindaaker. 2019. An Experimental Study of Context-Free Path Query Evaluation Methods. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management (SSDBM '19)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/3335783.3335791

[12] Ciro M. Medeiros, Martin A. Musicante, and Umberto S. Costa. 2018. Efficient Evaluation of Context-free Path Queries for Graph Databases. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. ACM, New York, NY, USA, 1230–1237. https://doi.org/10.1145/3167132.3167265

[13] Saskia Metzler and Pauli Miettinen. 2015. On Defining SPARQL with Boolean Tensor Algebra. *CoRR* abs/1503.00301 (2015). arXiv:1503.00301 http://arxiv.org/abs/1503.00301

[14] H. Miao and A. Deshpande. 2019. Understanding Data Science Lifecycle Provenance via Graph Segmentation and Summarization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1710–1713.

[15] Nikita Mishin, Iaroslav Sokolov, Egor Spirin, Vladimir Kutuev, Egor Nemchinov, Sergey Gorbatyuk, and Semyon Grigorev. 2019. Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication. In *Proceedings of the 2Nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'19)*. ACM, New York, NY, USA, Article 12, 5 pages. https://doi.org/10.1145/3327964.3328503

[16] Jakob Rehof and Manuel Fähndrich. 2001. Type-Base Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. *SIGPLAN Not.* 36, 3 (Jan. 2001), 54–66. https://doi.org/10.1145/373243.360208

[17] Fred C. Santos, Umberto S. Costa, and Martin A. Musicante. 2018. A Bottom-Up Algorithm for Answering Context-Free Path Queries in Graph Databases. In *Web Engineering*, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). Springer International Publishing, Cham, 225–233.

[18] Petteri Sevon and Lauri Eronen. 2008. Subgraph Queries by Context-free Grammars. *Journal of Integrative Bioinformatics* 5, 2 (2008), 157 – 172. https://doi.org/10.1515/jib-2008-100

[19] Arseniy Terekhov, Artyom Khoroshev, Rustam Azimov, and Semyon Grigorev. 2020. Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/3398682.3399163

[20] Ekaterina Verbitskaia, Semyon Grigorev, and Dmitry Avdyukhin. 2016. Relaxed Parsing of Regular Approximations of String-Embedded Languages. In *Perspectives of System Informatics*, Manuel Mazzara and Andrei Voronkov (Eds.). Springer International Publishing, Cham, 291–302.

[21] Ekaterina Verbitskaia, Ilya Kirillov, Ilya Nozkin, and Semyon Grigorev. 2018. Parser Combinators for Context-free Path Querying. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala (Scala 2018)*. ACM, New York, NY, USA, 13–23. https://doi.org/10.1145/3241653.3241655

[22] Mihalis Yannakakis. 1990. Graph-theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '90)*. ACM, New York, NY, USA, 230–242. https://doi.org/10.1145/298514.298576

[23] Xiaowang Zhang, Zhiyong Feng, Xin Wang, Guozheng Rao, and Wenrui Wu. 2016. Context-Free Path Queries on RDF Graphs. In *The Semantic Web – ISWC 2016*, Paul Groth, Elena Simperl, Alasdair Gray, Marta Sabou, Markus Krötzsch, Freddy Lecue, Fabian Flöck, and Yolanda Gil (Eds.). Springer International Publishing, Cham, 632–648.

[24] Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 197–208. https://doi.org/10.1145/1328438.1328464

Short Paper

Answer Graph: Factorization Matters in Large Graphs

Zahid Abul-Basher
University of Toronto
Toronto, Canada
zahid@mie.utoronto.ca

Nikolay Yakovets
Eindhoven University of Technology
Eindhoven, The Netherlands
hush@tue.nl

Parke Godfrey
York University
Toronto, Canada
godfrey@yorku.ca

Stanley Clark
Eindhoven University of Technology
Eindhoven, The Netherlands
s.clark@tue.nl

Mark Chignell
University of Toronto
Toronto, Canada
chignell@mie.utoronto.ca

ABSTRACT

Our *answer-graph method* to evaluate SPARQL *conjunctive queries* (CQs) finds a *factorized* answer set first, an *answer graph*, and then finds the embedding tuples from this. This approach can reduce greatly the cost to evaluate CQs. This affords us a second advantage: we can construct a cost-based planner. We present the answer-graph approach, and overview our prototype system, Wireframe. We then offer *proof of concept* via a micro-benchmark over the YAGO2s dataset with two prevalent *shapes* of queries, snowflake and diamond. We compare Wireframe's performance over these against PostgreSQL, Virtuoso, MonetDB, and Neo4J to illustrate the performance advantages of our answer-graph approach.

1 INTRODUCTION

Science is, of course, driven by observation. It is now becoming also ever more driven by data. Some of the datasets involved are unimaginably large. The data is often wildly heterogeneous, and rarely well structured as in business applications. This demands new skills, methods, and approaches of scientists, and challenges computer scientists with devising new data models, query languages, systems, and tools that better support this.

Graph-like data has become prevalent among scientific data stores and elsewhere. The data-science research community has begun to focus on how best to support the management of graph data and its analysis. One *data model* for graph databases is the *Resource Description Framework* (RDF) [18], paired with the *query language* SPARQL [8]. These have evolved as W3C standards, initially for addressing the Semantic Web. An RDF store conceptually consists of a set of *triples* to represent a directed, edge-labeled multi-graph. The triple $\langle s, p, o \rangle$ represents the directed, labeled edge from *subject* node "s" to *target* node "o" with *label* (*predicate*) "p". In RDF, nodes have unique identity. The semantics, however, is carried by the labels and how the nodes are connected. The UniProt [16] SPARQL Endpoint (dataset) [17], for example, consists of 63,376,853,475 RDF triples as of this writing. UniProt (Universal Protein resource) is a freely accessible, popular repository of protein data.

The SPARQL query language provides a formal way to query over such graph databases. Types of SPARQL queries can be thought about as small graphs themselves, so-called *query graphs*. In a SPARQL *conjunctive query* (CQ), the "nodes" are the query's *binding variables* and the "edges" between these are the labels

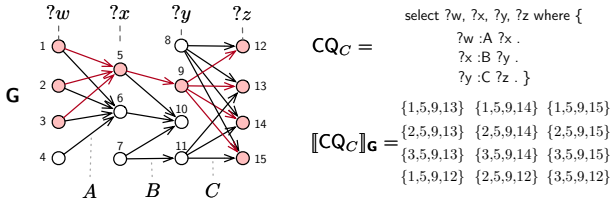


Figure 1: Example of an answer graph (shaded red).

to be matched. An "answer" to a CQ over a *data graph* **G** (that is, the graph database), denoted as $[\![\mathrm{CQ}]\!]_{\mathbf{G}}$, is a homomorphic *embedding* of the query graph into the data graph that matches the query's labels to the data graph's labeled edges. An answer is then a tuple of node ID's as a binding of the query's node variables. As such, each answer can be considered as a sub-graph matched in the data graph.

A CQ can be quite expensive to evaluate and may require extreme resources, given both the potentially immense size of the data graph and the relative complexity of the CQ's query graph. The challenge is to reduce the expense and needed machinery. We present a novel approach to *query optimization* and *evaluation* for CQs that we call Wireframe. A set of embeddings is the CQ's answer; this in itself is *not* a graph. In Wireframe's approach, as an intermediate step, we instead find the *answer graph*, the subset of edges from the data graph that suffices to compose the CQ's embeddings. This affords us powerful advantages: the answer graph is essentially the ideal *factorization* of the embeddings;[1] and we can find a best plan by estimated cost to evaluate this answer graph by *cost-based plan enumeration* via dynamic programming.

Wireframe's runtime evaluation employs two reduction mechanisms over the accumulating answer graph—*node burnback* and *edge burnback*—to guarantee a minimal factorized edge set. Given this evaluation paradigm, it is possible to devise a cost-based planner. Wireframe's optimization and evaluation is implementation agnostic; it can be easily implemented on any RDF-system architecture.

We presented the vision of Wireframe's approach in [9]. We have since developed and implemented the approach. Herein, we demonstrate Wireframe's key advantages via a prototype implementation, and compare its performance over a micro-benchmark against competing approaches.

[1] Wireframe can guarantee the minimum answer graph for *acyclic* CQs, and the minimum answer graph modulo the choice of triangulation of the CQ for *cyclic* CQs.

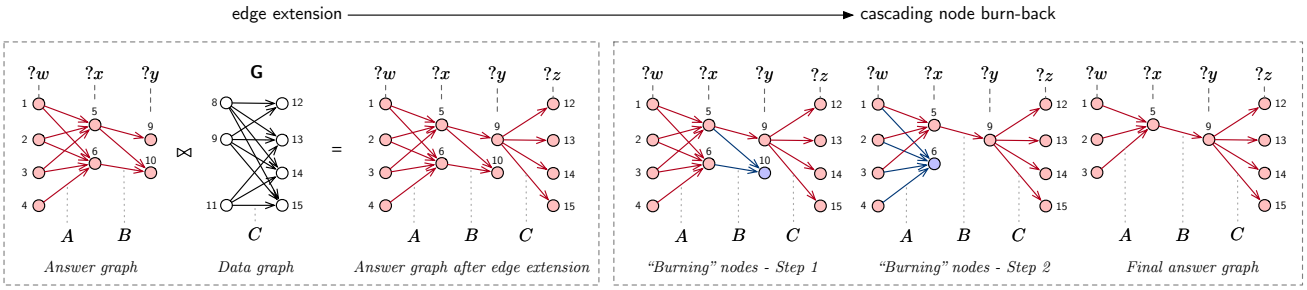


Figure 2: An example of the evaluation model for the answer graph generation.

2 RELATED WORK

RDF Systems. Over the past decade, a number of RDF systems have been developed. We can categorize them into: triple stores; property tables; column-based stores; and graph-based stores.

In triple stores, RDF data is stored in a long, but slim table with three columns where each row is a triple, $\langle s, p, o\rangle$, of RDF data [12]. For conjunctive queries (CQs), this single long, slim table approach requires self-joins over the table many times, which can lead to bad query performance. Large table scans and index look-ups can also lead to bad selectivity estimation, and, therefore, poor query optimization [1]. To overcome this, an index over the triple store for each of the six permutations over S, P, and O is often maintained [12, 24]. RDF-3x uses this approach by building a clustered B++ trees for each permutation of $\langle s, p, o\rangle$: $\langle s, p, o\rangle$, $\langle s, o, p\rangle$, $\langle p, s, o\rangle$, $\langle p, o, s\rangle$, $\langle o, s, p\rangle$, and $\langle o, p, s\rangle$ [12, 13]. It also maintains additional nine aggregate indexes to include the six binary and three unary projections of $\langle s, p, o\rangle$, which is useful for selectivity estimation. The aggregate indexes eliminate the need for expensive self-joins, therefore, improving query performance. TripleBit [26] constructs a compact triple matrix to minimize the use of indexes during query evaluation.

In contrast to triple stores, property tables use a flat but fat table where each row represents a subject value with columns for distinct property values [20]. In large, sparse RDF data, many cells of this single flat-fat table can be *null*, due to the absence of object values for the given subjects and predicates. To overcome this, Jena [20] clusters properties into different groups, and creates multiple property tables accordingly. BitMat [3] proposed a 3-D bit cube to represent subjects, predicates, and objects.

In column-based stores, RDF data is stored using multiple two-column tables [1]. There is a table for each unique property, with one column for the subject, and the other for the object. The tables can be stored using either physical row-store or column-store. This approach provides superior performance whenever there are value-based restrictions on properties. It can scale poorly, however, when the size of tables varies [15].

Graph-based stores are designed to handle graph manipulation over RDF data generally outside of the scope of SPARQL queries [6, 11]. These systems focus on specialized graph operations over RDF data [19]. For better performance, gStore [27] constructs VS-tree and VS*-tree index to evaluate both exact and wildcard SPARQL queries using subgraph matches. In [2], a compressed k^2-triples technique is used to run SPARQL queries in memory.

Factorization and Join Algorithms. Factorized databases are compact representations of relational tables [4]. They not only reduce the memory footprint while evaluating queries, but also reduce the query processing time by avoiding redundancy. This idea is even more effective for graph databases and queries, as we can show that our answer graph is an ideal factorization.

The *semijoin* operator can improve performance by ensuring everything in the *outer* (left) table *joins* with the *inner* (right) table. Wireframe's burnback mechanisms implement a form of semijoin, in a way. We likely could re-engineer our Wireframe burnback mechanisms via semijoin, were we to implement atop a platform providing a highly efficient semijoin.[2]

Worst-case optimal join algorithms use query decompositions for joins, accounting for the structural properties of the query along with the input relation statistics. This is in contrast to a traditional database where joins are evaluated "one join at a time" without taking the structure of the query into account [14].

Work that is related to ours in its mechanics is that of [25]. In [25], they reduce the transmission cost in distributed environments by generating a plan—*i.e.*, a sequence of semijoins—to evaluate acyclic conjunctive queries over datasets partitioned across different servers. Of course, our objectives in Wireframe and that of [25] are different, necessitating different methodologies. That said, our approach has advantages. Their algorithm for an acyclic CQ requires traversing the query tree *twice*. Wireframe does not need to. Their work does not apply to cyclic queries, whereas Wireframe does.

3 THE ANSWER-GRAPH APPROACH

In [4], the authors introduce the concept of *factorization* as a query-optimization technique for relational databases. Their technique is designed, and works exceptionally well, for schema and queries for which cross products of projections of the answer tuples all show up as answer tuples. This happens, for instance, in schema not in *fourth normal form*. Evaluating for these projected tuples first and then cross-producting them later can be a much more efficient strategy. Deciding how best to *factorize*—how to project into sub-tuples—is difficult, however.

For CQs, this last part is trivial: the factorization of the embedding tuples is fully down to component node pairs, corresponding to the labeled edges. This is our *answer graph*.[3] Factorization is *sometimes* a significant win for evaluating relational queries; it is *virtually always* a win for evaluating graph CQs.

An answer graph, AG, for a CQ is a subset of the data graph **G** that suffices to compute the embeddings for the CQ. We call the *minimum* such subset the *ideal answer graph*, iAG. The iAG is often quite small, significantly smaller than the set of embeddings, and extremely much smaller than **G**. Thus, we evaluate a CQ's embeddings in two steps: first, we find its iAG; then we compose

[2] This is future work of ours.

[3] This is demonstrably true when the CQ is *tree* shaped. This is arguable when the CQ has cycles. In the latter case, the factorization can be characterized as projections to tuples of node pairs and node triples (*triangles*).

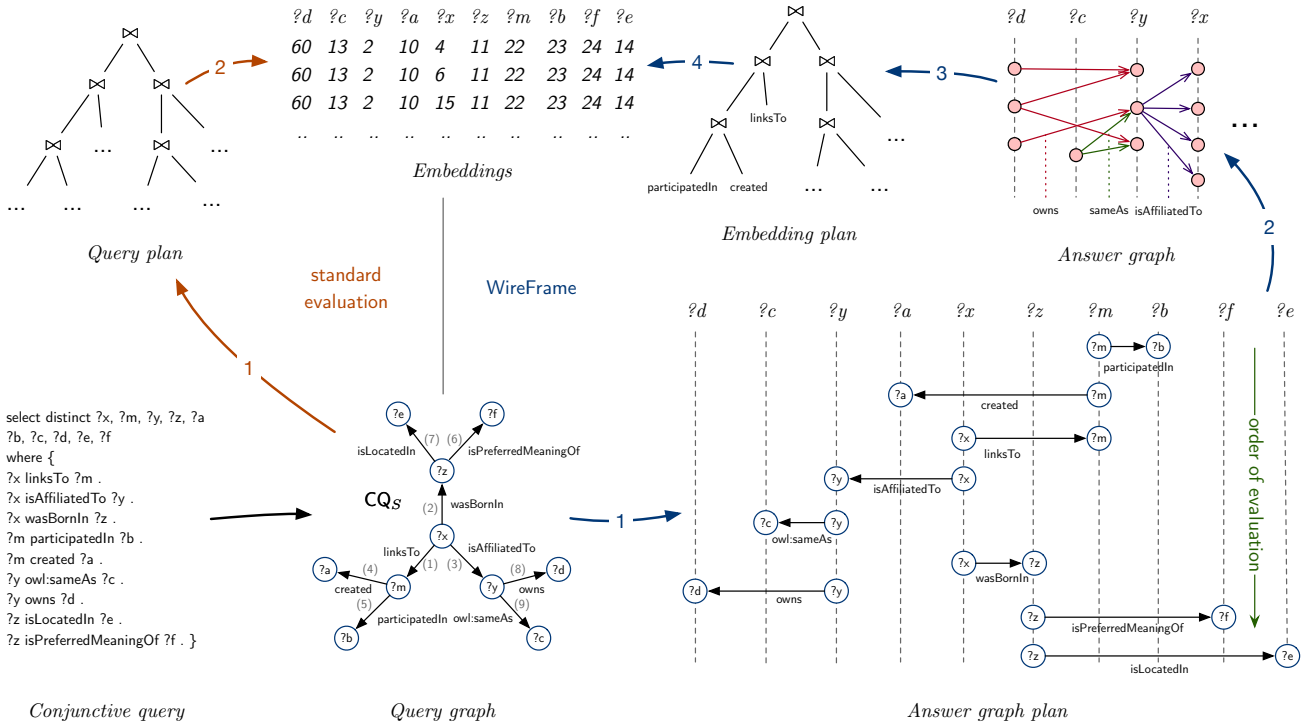


Figure 3: WIREFRAME: a two-phase cost-based optimizer for conjunctive queries.

the embeddings, which we call *defactorization*, from the iAG, rather than from **G**. This two-step approach can be significantly more efficient.

Consider the data graph **G** and the *chain* query CQ_C in Fig. 1. The query finds all tuples of nodes $\langle w, x, y, z\rangle$ from **G** such that $\langle w, x\rangle$ is connected by an edge labeled A, $\langle x, y\rangle$ by B, and $\langle y, z\rangle$ by C. Due to multiplicity from A-edges *fanning in* to, and C-edges *fanning out* of, B values, the embedding set is *twelve* tuples. Our answer graph consists of just *eight* labeled node pairs (in red). Such differences are greatly magnified when on a larger scale.

Our answer-graph approach affords us a second key advantage. We can devise a cost-based query optimizer based on dynamic programming to construct a *query plan*. A plan for us is simply a specified order of the CQ's query edges with which to evaluate to matching answer-graph edges. Our evaluation strategy for such plans is explained next, and our WIREFRAME optimizer for choosing plans is presented in Section 5.

4 THE EVALUATION MODEL

Our evaluation model for CQs then becomes two phase: *answer-graph generation* and *embedding generation*.

Answer-graph generation. For each query edge of the plan, in turn, our answer graph (AG) is populated with the matching labeled edges from **G** that meet the join constraints with the current state of the AG. Call this an *edge-extension* step. Then nodes in the AG that failed to extend are *removed*, and this "node burnback" cascades.

Consider the CQ with query edges $\langle ?w, A, ?x\rangle$, $\langle ?x, B, ?y\rangle$, and $\langle ?y, C, ?z\rangle$ in Fig. 2. Assume that the state of the AG after evaluating for query edges $\langle ?w, A, ?x\rangle$ and $\langle ?x, B, ?y\rangle$ is as shown in the figure, and that the next query edge to be evaluated is $\langle ?y, C, ?z\rangle$. This next edge is connected to the previously evaluated edges by the node variable $?y$. When retrieving data edges from **G** with label C (the query edge's label), one needs to ensure that the sources of retrieved data edges match to one of nodes bound to $?y$ in AG. After populating AG thusly, many nodes of $?y$ in AG may be "unattached" to any of the new edges; these nodes are marked to be removed during the node burnback procedure. In Fig. 2, the new answer graph has an unattached node, 10. During node burnback, this node is removed along with all of its edges, $\langle 5, 10\rangle$ and $\langle 6, 10\rangle$. Removing these edges can result in more unattached nodes, such as node 6, which no longer has any edge with the label A (contrasted with node 5). Thus, in the next iteration, node 6 is removed with all of its edges, $\langle 1, 6\rangle$, $\langle 2, 6\rangle$, $\langle 3, 6\rangle$, and $\langle 4, 6\rangle$. The node burnback procedure then terminates, as no further unattached nodes result.

Embedding generation. The embedding tuples are then generated over the answer graph by joining the answer edges appropriately. Given the *ideal* answer graph iAG and an acyclic CQ, the order in which we join is immaterial. No k-ary tuple is ever eliminated during a join with a next query edge from the iAG. This step is often quite fast, given the iAG is small. Evaluating this directly from the data graph **G**, on the other hand—which is what other evaluation methods for CQs do—can be exceedingly expensive. Fig. 3 illustrates, comparing a standard evaluation with our two-phase answer-graph approach.

5 THE PLANNERS

5.1 The Answer-Graph Planner

Plan Cost. The *edge walk* is our unit for estimating a plan's cost: the retrieval of a matching edge from **G**. To estimate the number of edge walks, *node* and *edge* cardinality estimations are made for each successive edge extension. Note that the cost of node burnback is amortised: every edge added that does not survive

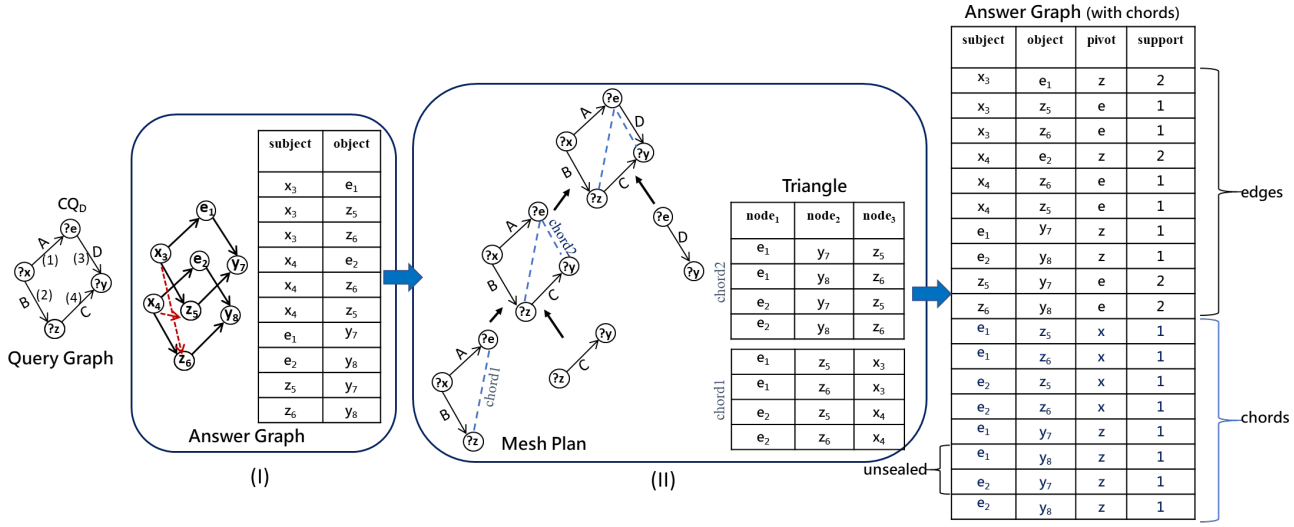


Figure 4: Triangulating a cyclic CQ using a mesh plan

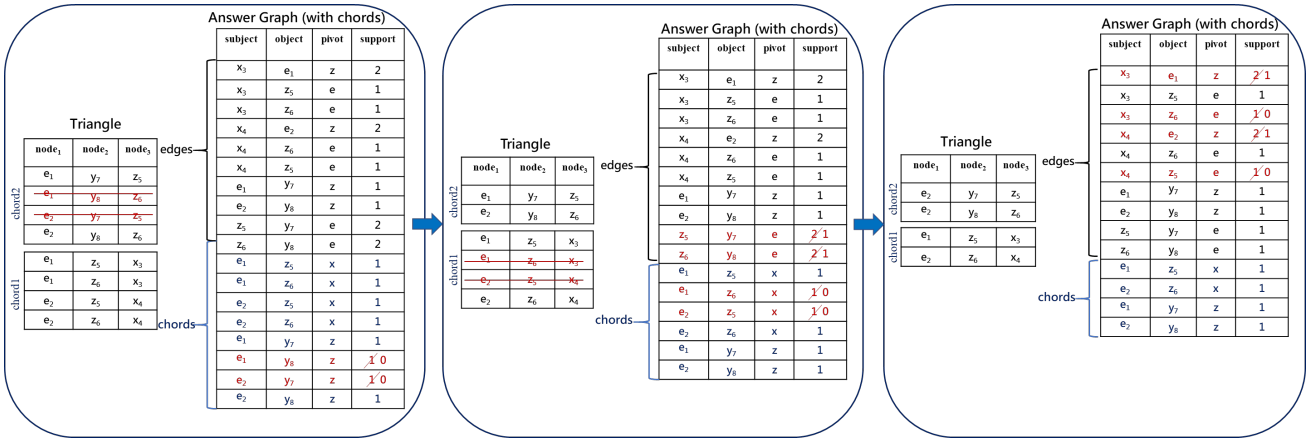


Figure 5: A running example of edge burnback procedure

to the iAG is, at some point, removed. WIREFRAME employs cardinality estimators drawn from a catalog consisting of 1-gram and 2-gram edge-label statistics computed offline [7, 10, 22, 23].

The Edgifier. A plan is a sequence of the CQ's query edges to be materialized. We employ a bottom-up, dynamic-programming algorithm to construct the edge order based on cost estimation (which relies upon the cardinality estimations).

When the query graph of a CQ has cycles—a *cyclic* query—there is an additional part to planning. Node burn-back suffices to generate the ideal answer graph for acyclic queries, but not for cyclic. The example in Fig. 4(I) illustrates why. *Spurious* edges (in red)—e.g., $\langle x_3, z_6 \rangle$ and $\langle x_4, z_5 \rangle$—can remain that do not participate in any final embeddings—i.e., $\langle x_3, z_5, y_7, e_1 \rangle$ and $\langle x_4, z_6, y_8, e_2 \rangle$. Nevertheless, one can still use this non-ideal answer graph to generate the embeddings using defactorization.

Even so, it is possible to reduce significantly further the answer graph. To cull spurious edges requires an *edge burnback* procedure in addition to node burnback. This requires the CQ's cycles have been *triangulated*; node *triples* are materialized in addition to the node *doubles* (the AG edges) during evaluation. Triangulation is the choice of which additional "query edges", which we call *chords*, to add.

The Triangulator. For cyclic CQs, in addition to the query-edge enumeration, cycles in the query graph of length greater than three are *triangulated* by adding *chord* edges. During evaluation, a chord is maintained as the intersection of the materialized joins of the opposite two edges for each triangle in which it participates. There are many different ways one can triangulate a CQ; the materialization cost depends on the order and choice of chord bisection of cycles (down to triangles). We employ a bottom-up dynamic programming algorithm to generate a bushy plan—we call this a *mesh plan*—that dictates such an order and choices.

The mesh plan, when executed with node burnback but not edge burnback, guarantees that the node sets, but not necessarily the edge sets, will always be minimal. A correct answer graph, AG, will be found, but it is no longer guaranteed to be ideal. as spurious edges may remain in the AG, as demonstrated in Fig. 4(I). The embeddings can, of course, be found from this non-ideal AG.

Edge Burnback. With the addition of *edge burnback* mechanism at runtime, we can guarantee again that we find the ideal AG (iAG) when answering cyclic CQs, *modulo* the choice of chords that were added. This works by checking the chords' materializations to chase what needs to be removed on cascade. This ensures that spurious edges are removed.

Fig. 4(II) shows a mesh plan which establishes as sequence of chords to triangulate the given query graph CQ_D. At the bottom of the plan, the first chord $\langle ?e, ?z\rangle$ is added, which creates the triangle $\bigtriangleup_{?e,?x,?z}$ by joining two query edges $\langle ?x, ?z\rangle$ and $\langle ?x, ?e\rangle$ on their common node $?x$. At the next step up, a second chord $\langle ?e, ?y\rangle$ is added to create a triangle $\bigtriangleup_{?e,?z,?y}$ by joining the previous chord $\langle ?e, ?z\rangle$ and the query edge $\langle ?z, ?y\rangle$ on their common node $?z$. At the root of the plan, a "seal" process occurs between the chord $\langle ?e, ?y\rangle$ of the left subplan and the query edge $\langle ?e, ?y\rangle$ of the right subplan, where it constructs the full cycle of the query graph. Each time we triangulate, we record the node triples (as triangles) drawn accordingly from **G** in the Triangle table, and the data chords in the Answer Graph table. We also record in the support column of the Answer Graph table the number of different triangles that each data edge (or data chord) participate, grouped by its opposite node label (which we call the *pivot*). For example, in the Answer Graph table, the edge $\langle x_3, e_1\rangle$ has a support of 2 with the pivot of z; that is, it is part of two triangles where the opposite node is of the label z. We can find these from the Triangle table: $\bigtriangleup_{e_1 z_5 x_3}$ and $\bigtriangleup_{e_1 z_6 x_3}$.

The seal process marks all uncommon data edges and data chords between the right and left subplans—we call these *unsealed* edges—by updating their supports in the Answer Graph table to 0. Unsealed edges represent the arcs that are not part of any cycle. Data edges that reside only on such arcs cannot be part of the ideal AG. The task of edge burnback is to remove these edges. It begins by removing all unsealed edges, along with triangles in which they participate. Removing a triangle involves removing all its data edges, which might also belong to other triangles in the query graph, thereby removing those triangles too. This process cascades until no unsupported triangle is left to be removed. Whenever we remove a triangle, we decrease the support of all of its constituent edges by one in the Answer Graph table. When the support of any data edge or chord in the Answer Graph becomes 0, then it is safe to remove it along with the triangles in which it participates. The process cascades until there are no edges with zero-support remaining in the Answer Graph table.

Fig. 5 demonstrates the process of edge burnback. After the sealing process, the unsealed set of edges are $\{\langle e_1, y_8\rangle, \langle e_2, y_7\rangle\}$, as shown in the Answer Graph table of Fig. 4(II). We next update the support of these two edges to 0 in the Answer Graph table. We then call the edge burnback procedure. This first removes the zero-support edges from the Answer Graph table. Next, this deletes triangles $\bigtriangleup_{e_1 y_8 z_6}$ and $\bigtriangleup_{e_2 y_7 z_5}$ which those edges participated from the Triangle table. The support for each of the edges in the Answer Graph table that was part of a recently deleted triangle is decremented by 1. For example, for the removed triangle $\bigtriangleup_{e_1 y_8 z_6}$, the support is decremented for edges $\langle z_6, y_8\rangle$ and $\langle e_1, z_6\rangle$. For the removed triangle $\bigtriangleup_{e_2 y_7 z_5}$, the support is decremented for edges $\langle e_2, z_5\rangle$ and $\langle z_5, y_7\rangle$. In the next iteration, the zero-support edges are removed from the Answer Graph table, which leads, in turn, to triangles $\bigtriangleup_{e_1 z_6 x_3}$ and $\bigtriangleup_{e_2 z_5 x_4}$ being deleted from the Triangle table. The support is then decremented for the edges that participated in those deleted triangles. And so forth. This process halts once there is no edge left with zero-support in the Answer Graph table. The resulting answer graph is then the ideal AG (iAG).

The overhead of edge burnback must be balanced off against the benefit of obtaining the iAG versus a larger, non-ideal AG. This is work in progress. In our experiments, our evaluation over cyclic CQs is without edge burnback.

5.2 The Embedding Planner

Plan Cost. When generating the embeddings for an acyclic CQ from its iAG, the order in which we join (connected) answer edges is immaterial. As the k-ary tuples are extended, no intermediate results are ever lost. Thus, for this, no planning is required.

The Defactorizer. On the other hand, when the CQ is cyclic, or when the AG provided is non-ideal, intermediate results can be lost. The join order then matters. We call this process *defactorization*. Alternative plans for embedding materialization are synonymous with choosing this join order. It is possible to do this again via a cost-based approach via a bottom-up, dynamic programming algorithm, using our catalog statistics.

6 EXPERIMENTS

Prototype. We have implemented a prototype, Wireframe, that runs on top of PostgreSQL, a popular relation database system. Wireframe implements the two phases described in Section 5, each with a separate planner and evaluator. The planner for the first phase outputs an optimal left-deep tree plan that indicates the execution order of the query edges. The evaluator then takes the tree plan to evaluate the query edges in sequence. For the second phase, we presently use a greedy approach to generate a tree plan based on the available statistics from the AG phase. The node burnback procedure is implemented via procedural SQL.

Environment. To evaluate Wireframe's performance, we use the YAGO2s dataset [21] containing 242M triples with 104 distinct predicates. With a select set of 10 acyclic and 10 cyclic CQs, we compare query execution times against PostgreSQL v11.0 (PG), Virtuoso v6.01 (VT), MonetDB v11.31 (MD), and Neo4J v.3.5 (NJ). All experiments were conducted on a server running Ubuntu 18.04 LTS with two Intel Xeon X5670 processors and 192GB of RAM.

Micro-benchmark. For the queries, we implemented a query miner that generates valid, non-empty queries over a dataset using query templates (with placeholders for edge labels). For our experiments, we use two templates, CQ_S and CQ_D, as shown in Figures 3 and 4, respectively. With these two templates, we mined 218,014 snowflake-shaped queries and 18,743 diamond-shaped queries. For our preliminary experimental study, we chose top ten queries in the size of final embeddings for each shape.

While in [5], it is argued that there are no use cases for cyclic queries, many, including us, have argued there certainly are. In [14], they discuss how triangle queries, the simplest of cyclic, have become increasingly popular for social networks, biological motifs, and graph databases. And that cyclic queries have not been used much yet in practice has been due in large part to that they have been too expensive to evaluate.

Comparators. For PostgreSQL and MonetDB, the dataset was imported as a triple store, with indexes on the string dictionary, and six composite indexes over the permutations of *subject*, *predicate*, and *object*. We set the size of the memory pool to eight GB for all of the systems, except for MonetDB (which sets its own resource allocations based on the server). We repeat execution of each query five times, taking the average of the last four runs (*i.e.,* warm cache), as reported in Table 1. The execution time is the time spent to retrieve all the result tuples for a query.

Results. One can observe from Table 1 that the size of the answer graph is exceedingly smaller than the number of embeddings. For instance, for the second snowflake-shaped query, the AG is 2,867 times smaller than the number of embeddings. It is no surprise, therefore, that Wireframe (WF) achieves good performance; it

CQ_S	**Snowflake-shaped Queries** (1/2/3/4/5/6/7/8/9)	PG	WF	VT	MD	NJ	|iAG|	|Embeddings|
1	diedIn/influences/actedIn/owns/wasCreatedOnDate/actedIn/created/hasDuration/wasCreatedOnDate	66	**4**	*	*	*	1660	2931986
2	hasChild/influences/actedIn/actedIn/wasBornIn/created/actedIn/hasDuration/wasCreatedOnDate	63	**3**	246	*	*	993	2847184
3	isCitizenOf/influences/actedIn/exports/wasCreatedOnDate/actedIn/created/hasDuration/wasCreatedOnDate	37	**7**	287	*	*	1140	2670339
4	isMarriedTo/influences/actedIn/actedIn/wasBornOnDate/created/actedIn/hasDuration/wasCreatedOnDate	59	**3**	286	*	*	3317	2569017
5	isMarriedTo/influences/actedIn/wasBornOnDate/isMarriedTo/actedIn/created/wasCreatedOnDate/hasDuration	57	**17**	268	*	*	3580	2127992
6	isMarriedTo/influences/actedIn/hasGender/isMarriedTo/actedIn/created/hasDuration/wasCreatedOnDate	57	**12**	268	*	*	3580	2123951
7	diedIn/isMarriedTo/actedIn/owns/wasCreatedOnDate/actedIn/created/hasDuration/wasCreatedOnDate	30	**15**	266	*	*	10761	2111948
8	isMarriedTo/influences/actedIn/hasFamilyName/isMarriedTo/created/actedIn/hasDuration/wasCreatedOnDate	32	**14**	261	*	*	3580	2102297
9	isMarriedTo/hasChild/actedIn/wroteMusicFor/created/created/actedIn/hasDuration/wasCreatedOnDate	35	**9**	256	*	*	7330	1786626
10	isMarriedTo/influences/actedIn/actedIn/created/created/directed/hasDuration/wasCreatedOnDate	39	**4**	237	*	*	3317	1533188
CQ_D	**Diamond-shaped Queries** (1/2/3/4)	PG	WF	VT	MD	NJ	|AG|	|Embeddings|
11	isLocatedIn/linksTo/isCitizenOf/livesIn	*	**39**	*	*	*	813311	59695937
12	livesIn/isCitizenOf/isLocatedIn/linksTo	*	**81**	*	*	*	833355	58785214
13	isCitizenOf/wasBornIn/linksTo/diedIn	*	**12**	*	*	297	132961	3141996
14	isCitizenOf/diedIn/linksTo/wasBornIn	*	**21**	*	*	296	251054	3124213
15	wasBornIn/isAffiliatedTo/linksTo/playsFor	*	**37**	*	*	*	470196	2310680
16	wasBornIn/playsFor/linksTo/isAffiliatedTo	*	**39**	*	*	*	471520	2299729
17	isConnectedTo/linksTo/extractionSource/byTransport	*	**33**	67	*	140	112040	1312372
18	created/rdfs:label/linksTo/isPreferredMeaningOf	*	264	**65**	203	130	772994	169380
19	linksTo/isPreferredMeaningOf/created/skos:prefLabel	*	114	**22**	111	135	766785	169324
20	diedIn/linksTo/wasBornIn/graduatedFrom	*	**12**	92	*	195	68720	106214

Table 1: Query execution time (sec) in different systems (* denotes terminating the query after 300 seconds).

avoids the redundant edge-walks that arise from many-many joins. While the second snowflake-shaped query took 63 seconds on PostgreSQL, it only took three seconds on Wireframe. The AG approach requires a much smaller memory footprint, which can be beneficial for traditional database systems that heavily use secondary storage. The approach also competes well against main-memory intense systems such as Neo4J and Virtuoso. For the cyclic, diamond-shaped queries, employing only node burnback does not guarantee the ideal answer graph, as discussed above. We have found that the resulting AGs can be significantly larger than the ideal, sometimes close to the number of embeddings. For this reason, Wireframe was slower for some of the cyclic queries, notably 18 and 19. Even so, its performance over cyclic queries is quite good. With further plan- and run-time optimization with edge burnback, we believe that the performance will be stellar. One can view our approach as an additional optimization technique on top of traditional databases to handle SPARQL CQs or to boost the performance of existing RDF systems.

7 CONCLUSIONS

We have clear objectives for our next steps. *First*, one has a richer plan space when considering bushy plans for both our first and second phases. The challenge is to devise a suitable cost model for searching the bushy-plan space via dynamic programming. *Second*, when the size of an answer graph is distant from the ideal, generating the embeddings can be costly. Triangulation promises to reduce this significantly. This requires investigating the trade-offs between the added cost for maintaining the triangle materializations and the reduced cost from generating the embeddings from the significantly smaller ideal AG. *Lastly*, we are to explore further optimizations within this space. Large graphs are meant to be queried.

REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. VLDB '07, pages 411–422. VLDB Endowment, 2007.

[2] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, and M. A. Martínez-Prieto. Compressed k2-triples for full-in-memory rdf engines. *arXiv preprint arXiv:1105.4004*, 2011.

[3] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. WWW '10, pages 41–50, New York, NY, USA, 2010. ACM.

[4] N. Bakibayev, D. Olteanu, and J. Závodnỳ. FDB: A query engine for factorised relational databases. *VLDB*, 5(11):1232–1243, 2012.

[5] A. Bonifati, W. Martens, and T. Timm. An analytical study of large sparql query logs. *The VLDB Journal*, 29(2):655–679, 2020.

[6] V. Bonstrom, A. Hinze, and H. Schweppe. Storing rdf as a graph. In *Proceedings of the IEEE/LEOS 3rd International Conference on Numerical Simulation of Semiconductor Optoelectronic Devices*, pages 27–36, 2003.

[7] S. Christodoulakis. *On the estimation and use of selectivities in database performance evaluation.* CS Dept., U. of Waterloo, 1989.

[8] A. S. Eric Prud'hommeaux. SPARQL query language for RDF. W3C recommendation, 15 january, 2008.

[9] P. Godfrey, N. Yakovets, Z. Abul-Basher, and M. H. Chignell. Wireframe: Two-phase, cost-based optimization for conjunctive regular path queries. In *AMW*, 2017.

[10] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys (CSUR)*, 20(3):191–221, 1988.

[11] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. A path-based relational rdf database. In *Proceedings of the 16th Australasian Database Conference - Volume 39*, ADC '05, pages 95–103, Darlinghurst, Australia, Australia, 2005.

[12] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

[13] T. Neumann and G. Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases. *Proc. VLDB Endow.*, 3(1-2):256–263, Sept. 2010.

[14] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.

[15] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: Not all swans are white. *Proc. VLDB Endow.*, 1(2):1553–1563, Aug. 2008.

[16] The UniProt consortium (author notes), UniProt: a worldwide hub of protein knowledge. *Nucleic Acids Research*, 47(D1):D506–D515, January 2019.

[17] UniProt SPARQL endpoint. https://sparql.uniprot.org/,2020.

[18] W3C: Resource description framework (rdf). http://www.w3.org/TR/rdf-concepts/,2004.

[19] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, Aug. 2008.

[20] K. Wilkinson and K. Wilkinson. Jena property table implementation, 2006.

[21] YAGO2s: A high-quality knowledge base. http://yago-knowledge.org/resource/. Max Planck Institut Informatik.

[22] N. Yakovets. *Optimization of Regular Path Queries in Graph Databases.* PhD thesis, York University, 2016.

[23] N. Yakovets, P. Godfrey, and J. Gryz. Query planning for evaluating SPARQL property paths. In *SIGMOD*, pages 1–15. ACM, June 2016.

[24] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficiently querying rdf data in triple stores. In *Proceedings of the 17th international conference on World Wide Web*, pages 1053–1054. ACM, 2008.

[25] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

[26] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: a fast and compact system for large scale rdf data. *Proceedings of the VLDB Endowment*, 6(7):517–528, 2013.

[27] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: Answering sparql queries via subgraph matching. *Proc. VLDB Endow.*, 4(8):482–493, May 2011.

Short Paper

Schema Inference for Property Graphs

Hanâ Lbath
Lyon 1 University, ENS Lyon
France
hana.lbath@ens-lyon.fr

Angela Bonifati
Lyon 1 University
France
angela.bonifati@univ-lyon1.fr

Russ Harmer
CNRS, ENS Lyon
France
russ.harmer@ens-lyon.fr

Abstract

Graphs are pervasive in many applications in which interconnected data are used to represent, explore and predict digital and real-world phenomena. Oftentimes, graph data comes without a predefined structure and in a constraint-less fashion, thus leading to inconsistency and poor quality. In this paper, we present a novel end-to-end schema inference method for property graph schemas that tackles complex and nested property values, multi-labeled nodes and node hierarchies. Our method consists of three main steps, the first of which builds upon Cypher queries to extract the node and edge serialization of a property graph. The second step builds over a MapReduce type inference system, working on the serialized output obtained during the first step. The third step analyzes subtypes and supertypes to infer node hierarchies. We present our schema inference pipeline under two variants, namely a label- and a property-oriented variant. Finally, we experimentally evaluate and compare its scalability and accuracy on several real-life datasets. To the best of our knowledge, our work is the first to address schema inference for property graphs.

1 Introduction

Over the past decade, graph-based knowledge representation has been becoming increasingly popular, be it with the advent of graph databases [4] as an alternative to relational databases, or to model complex systems, from social and fraud detection networks to smart city grids or neuronal networks. Graphs are applicable to all settings in which interconnected data are used to represent, explore and predict digital and real phenomena. Understanding the connections in the data is fundamental for companies to carry out analytical and machine learning tasks.

Oftentimes, graph data comes without a structure and in a constraint-less fashion, thus leading to inconsistency and poor quality. According to a recent survey [12], the most recurrent task for users of real-world graphs is data integration. As a matter of fact, graphs naturally lend themselves to information reconciliation and integration. However, the integration of large-scale graphs, e.g. in knowledge bases such as DBPedia, Wikipedia or the more recent CovidGraph, might turn out to be incorrect and error-prone if not guided by means of schema constraints. These constraints are also important pillars for query optimization and metadata management, the latter being unexplored topics in graph databases. There are several models for representing graphs. Among the most popular is the property graph (PG) data model, which is a multigraph with both labeled nodes and edges, along with property value pairs associated to both. It has gained adoption with systems such as ArangoDB, HANA Graph, Neo4j, Oracle PGX, TigerGraph, Titan, etc.

However, due to the lack of a standard schema, PG instances are typically built without a predefined schema. Although it ensures great flexibility, it can also become a great impediment, notably whenever the structure of the underlying instance has to be stabilized as in many data management settings. Indeed, schemas succinctly represent the structure of the PG instances and allow to set constraints, such as the types of nodes, edges and properties as well as the cardinality of a relationship or property value data types. Moreover, schemas can be used to build relevant graph features based on types as needed in many Machine Learning pipelines [3]. Yet, given that PG instances usually exist prior to the schema definition, extracting a schema from those instances in a principled way might become a non-trivial and challenging task. Existing PG schema inference methods in available graph databases (such as Neo4j) are simplistic in that they can only output basic edge types and node types and do not take into account the complexity of the PG data model. In particular, they cannot handle complex data types, overlapping node types or node hierarchies. A principled approach to PG schema inference is currently missing and highly desirable given the interest in an ongoing ISO SC32/ WG3 standardization process of PG schemas, involving people from academia and industry in the LDBC community[1].

In this work, we address this problem and present a novel end-to-end schema inference method covering the entire spectrum of features of the PG data model. Our method tackles complex and nested property values, multi-labeled and unlabeled nodes, node hierarchies and overlapping node types as well as edge cardinality constraints and optionality of properties. We also introduce two variants of our method, a label-oriented and a property-oriented one and investigate their pros and cons.

To enable scalability, our method leverages a MapReduce approach [2] developed for schema inference from JSON datasets, which both aggregates types and identifies data types. However, the JSON data model and the PG data model are significantly different. While the former can be seen as an edge-labeled tree-structured data model, the latter is more expressive as it is a multi-graph with novel schema components such as subtyping and edge cardinality constraints. Because of that, schema inference for property graph is a challenging and non-trivial problem that we tackle in this paper for the first time.

The schema inference pipeline we designed[2] can be divided into three main steps. First, we employ Cypher queries to extract and serialize the nodes and edges of the input PG, in addition to gathering information needed to infer edge cardinality constraints. Cypher[3] is an open-source graph query language developed by Neo4j, inspired from SQL and adopted by several graph database vendors. Afterward, we infer node and edge types, together with the property value data types, using the output from the first step to input the MapReduce algorithm. The last step consists in analyzing subtypes and supertypes to infer node

[1] More information can be found at: https://www.gqlstandards.org
[2] The source code is available at: https://gitlab.com/Hgit/pgsinference
[3] http://www.opencypher.org

hierarchies. Our schema inference method is generic and can be adapted to other graph database platforms, insofar as the input PG can be appropriately serialized to JSON. Furthermore, our work can serve as a basis to inform the ongoing discussion of the working groups within the ISO SC32/ WG3 standardization process about the impact of the schema inference process on the PG schema design choices. After describing our schema inference method, we experimentally evaluate the accuracy and scalability of our schema inference method on real-life datasets.

2 Related Work

Several vendors propose graph databases supporting the PG data model, such as Neo4j, Oracle PGX, TigerGraph, RedisGraph and GraphQL. Neo4j offers a limited possibility to view a schema of the database via a Cypher query that outputs a single-labeled directed graph displaying which types of nodes can be connected together and through which types of edges. However, there is a lack of support of the PG data model in its full potential [4]. In particular, multi-labeled nodes in the graph instance are duplicated in the graph schema so that each node is assigned a single label, hence loosing label co-occurrence information, which is a crucial capability of the PG data model. Furthermore, properties, edge cardinality constraints and node type hierarchies are disregarded. Nonetheless, other Cypher queries output for each node (or edge) type its corresponding properties and their data types, in addition to whether or not they are mandatory. Nevertheless, in the case of multi-valued or nested properties, only the data type of the data structure containing them is inferred. In addition, GraphQL schemas can be inferred from Neo4j databases [7] via a Neo4j Desktop GraphQL plugin or neo4j-graphql-js. Node and edge types and node properties data types are inferred, unlike overlapping types, node hierarchies and nested property values—in contrast with our method. Furthermore, some Neo4j-specific data types, such as Locations or Dates, produce an error.

Many schema inference approaches consist in identifying structural graph summaries by grouping *equivalent* nodes together [10]. Typical examples are clustering techniques, like [9] and [6], which infer *types* in RDF datasets. Both are based on the assumption that the more *properties* two *entities* share, the likelier they belong to the same *type*. To this end, they group entities according to a similarity metric. They also both handle hierarchical and overlapping types. In [6], a density-based clustering method, DBSCAN, is adopted. In [9] a faster and more accurate clustering method, called StaTIX, is proposed. It uses the cosine similarity metric and is based on community detection. However, none of these techniques are suitable for PGs.

In [2], the authors propose a scalable MapReduce approach for schema inference in JSON datasets, which infers all data types before merging types according to an equivalence relation. Our pipeline repurposes it for PG type inference. However, in JSON, type hierarchies only exist in a very limited capacity and [2] does not tackle explicitly the problem of overlapping types. Due to the remarkable differences between the JSON data model and the PG data model, their method is not directly applicable to PGs.

In summary, none of the above approaches fully satisfy our criteria for PG schema inference: inference of types, basic and complex data types, overlapping types and node hierarchies, and, to the best of our knowledge, ours is the first work presenting a schema inference method specifically tailored to PGs.

3 Preliminaries

In this section, we recall the definition of property graphs [1, 4, 5] and extend a PG schema definition to best fit the schema inference process presented in this paper.

Let $\mathcal{O}$ be a set of *objects*, $\mathcal{L}$ be a finite set of *labels*, $\mathcal{K}$ be a set of *property keys*, and $\mathcal{N}$ be a set of *values*. We assume these sets to be pairwise disjoint.

Definition 3.1. A *property graph* is a structure $(V, E, \eta, \lambda, \nu)$ where

- $V \subseteq \mathcal{O}$ is a finite set of objects, called vertices;
- $E \subseteq \mathcal{O}$ is a finite set of objects, called edges;
- $\eta : E \to V \times V$ is a function assigning to each edge an ordered pair of vertices;
- $\lambda : V \cup E \to \mathcal{P}(\mathcal{L})$ is a function assigning to each object a finite set of labels (i.e., $\mathcal{P}(S)$ denotes the set of finite subsets of set S); and,
- $\nu : (V \cup E) \times \mathcal{K} \to \mathcal{N}$ is a partial function assigning values for properties to objects;

such that $V \cap E = \emptyset$ and the domain of ν is finite.

In [5], a Data Definition Language (DDL) for PGs is proposed where the obtained schema is a PG itself. We expand it to introduce *edge cardinality*, *subtypes* and *supertypes* and the concept of *inheritance edge type*. All these are key concepts to make our inferred PG schemas as expressive and accurate as possible.

Definition 3.2. (**Property Graph Schema**) A Property Graph Schema is a Property Graph Type, which is a triple $(\mathcal{BT}, \mathcal{NT}, \mathcal{ET})$ with $\mathcal{BT}$ a set of element types, $\mathcal{NT}$ a set of *node types*, $\mathcal{ET}$ a set of *edge types*.

- **Property type**: A property type is a pair (k, t), where $k \in \mathcal{K}$ is the property key and $t \in \mathcal{T}$ is its data type.
- **Element type**: An element type $b \in \mathcal{BT}$ is a quadruple (l, P, M, E), where $l \in \mathcal{L}$ is a label, P is a set of property types, $M \subseteq P$ is a subset of mandatory property types and $E \subseteq \mathcal{BT}$ is the set of element types that b extends.
- **Subtypes**: A subtype is an element type such that it inherits from another element type—called the **supertype**.
- **Node Type**: A node type is a pair (b, H), with $b \in \mathcal{BT}$, $H \subseteq \mathcal{BT}$ the set of **supertypes** b inherit from.
- **Inheritance Edge Type**: An inheritance edge type is a triple (s, e, t), where $s = (b, H) \in \mathcal{NT}$, $t \in H$, $e \in \mathcal{BT}$ with a label that we denote "SubtypeOf". Inheritance edge types do not have any cardinality.
- **Ordinary Edge Type**: An ordinary edge type is a quadruple (s, e, t, c), with $s \in \mathcal{NT}$ the source node, $t \in \mathcal{NT}$ the target node, $e \in \mathcal{BT}$, $c = ((i, k), (j, l)) \in (\{0, 1\} \times \{1, N\})^2$ the **cardinality**.
- **Edge Type**: the disjoint union of ordinary edge and inheritance edge types.

In the remainder of the paper, unless stated otherwise, *edge type* refers to *ordinary edge type*. Let us define overlapping types:

Definition 3.3. (**Overlapping Type**) An overlapping type is an element type which is a subtype of two or more supertypes.

4 Inferring Property Graph Schemas

In this section we present our method to infer a PG schema. We assume all nodes and edges in the PG are labeled. Nodes are of the same type if and only if they have the same set of labels, while edges are of the same type if and only if they share their set of source nodes, target nodes and edge labels. These assumptions

may be too strong in some cases. We will present in Section 4.4 an alternative that deals with some of its shortcomings.

Our PG schema inference pipeline can be divided into three main steps (cf. Fig. 1).

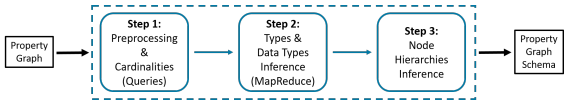


Figure 1: Schema inference main steps.

4.1 Step 1: Preprocessing

To begin with, the input PG needs to be serialized into JSON—the format required by the MapReduce algorithm. To this end, the graph is queried to match nodes and edges. To infer edge cardinality constraints, statistics are also collected. Then, the matched nodes and edges are serialized in such a way that proper node and edge type inference is guaranteed. From there, they are stored in `jsonline` files that will be input into the next step.

4.1.1 Preprocessing Queries. First, nodes are matched according to their labels via Cypher queries. Similarly, edges are matched according to their source, target node and edge labels. Node and edge types such that none of their instances have properties are stored separately. As there is no need to infer their property data types, the MapReduce step can be skipped in these cases and they can be directly processed by the third step.

4.1.2 Edge Cardinalities. Edge cardinality constraints are then inferred using rules comparing the number of instances of the source nodes, target nodes and a given edge type. These are collected by the preprocessing queries. Let us denote this edge type $E = (s, e, t, c)$, with $s, t \in \mathcal{NT}$, $e \in \mathcal{BT}$ and c the cardinality. For instance, if there are more target nodes than source nodes and there are as many target nodes as edges of type E, then the cardinality of E is *one-to-many*. This means that an instance of the source node type s can be linked to many instances of the target node type t via the edge type E but that an instance of t can only be linked to one instance of s via E. These rules can be similarly declined for *one-to-one*, *many-to-one* and *many-to-many* relationships. The cardinality constraints can be further refined to take into account optional relationships. For the given edge type E, if there are fewer instances of source nodes than of its corresponding node type, s, then this edge type is *optional* for the source node type s (otherwise, the edge type is *mandatory* for s). This means that there may be nodes of type s that are not linked to nodes of type t via an edge of type E. The same rule can be applied for target nodes.

In our pipeline, the cardinality constraint information is stored as an edge property with the key `meta_cardinality` and a `string` data type (e.g., `meta_cardinality` : "0..1:1..*" encodes a *one-to-many* relationship with the edge type *optional* for the source node type and *mandatory* for the target node type).

4.1.3 Serialization to JSON. The Cypher queries output node and edge `neo4j` objects where property values are sometimes incorrectly stored (e.g., a dictionary as a `string`). They are identified and converted accordingly to ensure a correct data type inference. Nodes and edges are then converted to dictionaries that are stored in `jsonline` files in such a way that correct type inference by the MapReduce method is guaranteed (cf. Section 4.2). The nodes file contains a single dictionary where each key-value pair represents a node. The key corresponds to its labels, sorted in alphabetical order and separated by colons. The value is a dictionary storing the properties as key-value pairs. Similarly, the edges file contains a single dictionary where each key-value pair represents an edge, with the value a dictionary storing the properties as key-value pairs. This time, the key corresponds to its starting node labels, its own labels and its target node labels, all separated by colons.

4.1.4 Example. Let us set a PG instance G of a social network of patients and doctors who can create and like posts and comments as well as reply to them. This is partially inspired by the LDBC Social Network Benchmark database [8]. Listings 1 and 2 are dictionaries encoding a node and edge instance.

```
{'Patient:Person': {
    'name': 'Alice',
    'birthday': {'day':29,
                 'month':'May',
                 'year':2000},
    'StudentNumber': 42,
    'address': ['Market Street', 'Lyon'] }}
```

Listing 1: Dictionary storing a node instance.

```
{'Patient:Person::KNOWS::Doctor:Person':
    {'date': '1993-06-02' }}
```

Listing 2: Dictionary storing an edge instance.

At the end of Step 1, we have two `jsonline` files ready to be input into the MapReduce algorithm, a list of node types with no properties and a list of all edge types containing edge cardinality information but no properties (they will be added in Step 2).

4.2 Step 2: Types and Data Types Inference (MapReduce)

In this step, we aggregate nodes and edges by type and infer the property data types by relying on MapReduce [2]. It can be summarized in two steps : i) a **Map** phase where all property value data types are inferred (cf. Listing 3) and ii) a **Reduce** phase where types are fused according to an equivalence relation. Here, we use the *kind-equivalence* relation described in [2] (cf. Listing 4). It fuses recursively types of the same *kind*, i.e. records with records, arrays with arrays and basic types (`String`, `Number` and `Boolean`) with basic types. The fusion of two basic types produces their union. The fusion of arrays outputs an array containing the fusion of their content. In the case of records, the different values of the two records are fused if and only if they share the same key. If one of the records contains keys that are not present in the other, then this particular key-value pair is deemed optional. Therefore, nodes will have their properties merged if and only if they share the same set of labels. Additionally, edges will have their properties merged if and only if they share the same set of source node, target node and arc labels. This complies with our assumption stated at the beginning of Section 4. The fusion function is recursively called so as to handle nested values.

The output of the algorithm, which is a JSON record, is then parsed and stored in a human-readable dictionary where question marks are affixed to optional properties' data types. A property "`meta_mandatory` : False" is instead added to optional records. Next, the dictionary is merged with the node types with no properties and the list of edge types containing cardinality constraints. Thus, the output of this step is a preliminary PG schema of the input PG, but it is still missing subtyping information.

4.2.1 Example. Listing 3 and 4 illustrate the fusion of two {`Person`, `Patient`} nodes using the *kind-equivalence* detailed above.

```
{'Patient:Person': {
  'name': STRING,
  'birthday':{'day': NUMBER,
    'month': STRING,
     'year': NUMBER},
  'StudentNumber': NUMBER
  'address': [STRING] }}
{'Patient:Person': {
  'name': STRING,
  'address': [NUMBER] }}
```

Listing 3: Two JSON record types corresponding to two nodes present in G.

```
{'Patient:Person': {
  'name': STRING,
  'birthday':
    {'day': NUMBER,
      'month': STRING,
      'year': NUMBER,
      'meta_mandatory': FALSE},
  'address': [NUMBER + STRING],
  'StudentNumber': NUMBER ? }}
```

Listing 4: Fusion of the two JSON record types on the left-hand side using the *kind-equivalence*.

4.3 Step 3: Inference of Node Hierarchies

The final step is to infer node type hierarchies so as to obtain a schema satisfying Definition 3.2. Inferring edge hierarchies is unnecessary in Neo4j graphs, since edges can only be associated with a single label. Nevertheless, we expect our hierarchy inference technique to straightforwardly extend to edge hierarchies.

Algorithm 1: Node Hierarchy Inference (Label-Oriented Variant)

```
   input : schema nodes and edges dictionaries from Step 2
   output: updated schema nodes and edges dictionaries and schema file
1                                                  ▷ Identify supertypes
2  supertypes = list of the pairwise intersections of the node label sets
3  for stype in supertypes do
4      add stype to nodes if needed
5                                                  ▷ Identify subtypes
6  nodeLabels = list of node label sets
7  for i ← 0 to length(nodeLabels) − 1 do
8      nset0 = nodeLabels[i]
9      for j ← 0 to length(nodeLabels[i :]) − 1 do
10                                  ▷ Two given node types are compared only once
11         nset1 = nodeLabels[j]
12         if nset0 ≠ nset1 then
13             if nset0 ⊂ nset1 then
14                 add nset1::SubtypeOf::nset0 edge
15             else if nset1 ⊂ nset0 then
16                 add nset0::SubtypeOf::nset1 edge
```

First, node **supertypes** corresponding to subsets of labels present in two or more node type label sets are inferred (l. 1-4). To this end, we take the pairwise intersection of the label sets of all node types inferred during Step 2. For instance, let us consider a {`Person`, `Doctor`} and a {`Person`, `Patient`} node type. The node type labeled {`Person`} is thus identified as a *supertype*.

The second step is to identify all **subtypes** (l. 5-16). We assumed earlier that node types are characterized by their labels. We thus consider a node type (with label set A) to be a *subtype* of a distinct node type (with label set B) if $B \subsetneq A$. This enables us to automatically handle overlapping types and hierarchies of arbitrary depths, as long as they are reflected in the labels. Thus, the label sets of all node types, including those inferred previously, are compared in pairwise fashion to identify subtypes. For example, {`Person`, `Patient`, `Doctor`} is a *subtype* of {`Person`, `Doctor`}. The corresponding inheritance edges are then created and added to the schema. The time complexity of the node hierarchy inference is quadratic in the number of node types inferred in the previous steps. However, since the number of node types is typically smaller than the size of the PG, this complexity remains bearable in practice, as also shown by our evaluation.

With the node type hierarchy inferred, the PG schema is complete. It is stored in a JSON file using the format described earlier.

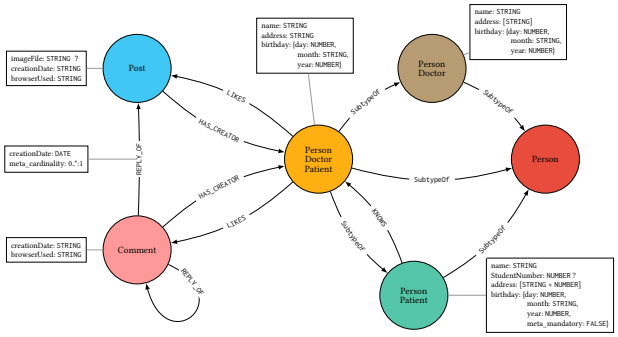


Figure 2: An excerpt of the PG schema inferred from the PG G using our label-oriented variant.

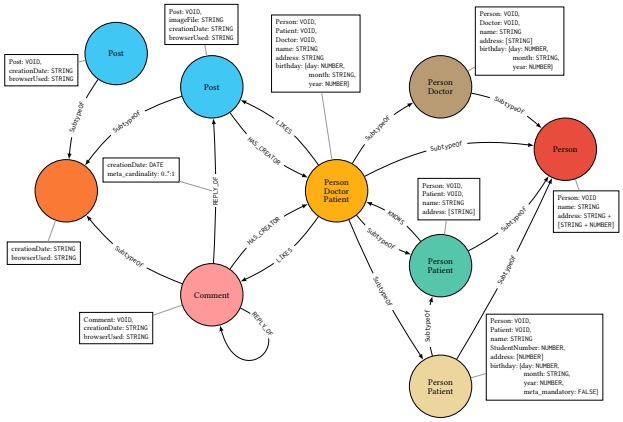


Figure 3: An excerpt of the PG schema inferred from the PG G using our property-oriented variant.

4.4 Method Variant: Labels as Properties

So far, we have assumed that all nodes in the input PG are labeled and that node labels characterize node types. However, these assumptions may sometimes be unsuitable. Indeed, some graphs may contain unlabeled nodes and information provided by the properties may be lost when only taking labels into account. For example, let us consider the PG G (cf. Example 4.1.4) and assume that the nodes labeled {`Person`, `Patient`} can be partitioned into two groups: those with a `StudentNumber` property key, corresponding to patients who are students, and those without one. With the previous approach, we had missed this subtlety and only identified a single {`Person`, `Patient`} node type with an optional `StudentNumber` key (cf. Listings 3 and 4 and Fig. 2). Therefore, we propose to consider labels as properties with a `Void` data type and to use property key sets (which now include labels) instead of label sets to characterize node types and thus identify node types and hierarchies. To merge nodes, we hence use $\mathcal{L}$*-driven reduction* [2], which fuses two records if and only if they share the same property key sets. As a result, no optional property can be inferred but rather property key co-occurrence information is identified. Moreover, unlabeled nodes can henceforth be considered on the same level as labeled nodes. In Neo4j, edges must have exactly one label. So, all the input PGs we considered contain single-labeled edges. We thus continue to utilize our label-oriented approach to handle them. If the source node or

target node is unlabeled, it is referred to by its property keys in place of labels. Our PG schema inference pipeline in this paradigm is very similar to the former and can be divided into the same three steps.

4.4.1 Example. Fig. 2 and 3 depict the schemas inferred from the PG G using both variants. As discussed above, the first approach overlooks the information provided by the properties, resulting in missing node types (e.g., a supertype of the `Post` and `Comment` nodes could not be inferred). This is resolved with the property-oriented variant where an unlabeled supertype could be inferred (the unlabeled orange node in Fig. 3).

5 Evaluation

Our method is implemented in Python 3 and is based on Neo4j 3.5. The graphs are queried with Cypher through the Neo4j Python driver. The MapReduce step is based on the implementation in [2], which runs with Spark 2.4.5. All experiments were performed on an Openstack Virtual Machine with twelve 2GHz 64-bits Intel Xeon CPUs, 62 GB of memory and a 1.5 TB hard drive.

5.1 Datasets and Metrics

We have used several datasets in our experimental study (cf. Table 1). We evaluated our schema inference method on the **LDBC Social Network Benchmark** (**LDBC**) [8], a synthetic social network. It contains single-labeled nodes and comes equipped with a ground truth schema incorporating node hierarchies. We also used two **Neuprint** datasets, corresponding to neuronal networks of different parts of the fruit fly brain: i) the mushroombody (**mb6**) [14] and ii) the medulla (**fib25**) [13]. Accompanied by a ground truth schema, they contain multi-labeled nodes (as opposed to LDBC) and a large diversity of property value data types, such as JSON records or `neo4j` cartesian 3D points. We tested as well our pipeline on a **Covid-19 graph** (**covid19**) (https://covidgraph.org/). This graph is being assembled by the CovidGraph project, which is currently ongoing. The graph is continuously evolving and hence so are the corresponding schemas. The results presented in this paper are those obtained with the April 2020 version, which notably holds multi-labeled nodes and five unlabeled nodes. No ground truth is available for the schema of the latter graph.

To assess the quality of our schema inference, we have used the precision, recall and F1-score of the node types and edge types. We consider an inferred type that is (not) present in the ground truth schema as a True Positive (TP) (False Positive (FP), respectively). A type that is present in the ground truth but not in the inferred schema is considered as a False Negative (FN). Precision accounts for the proportion of identified types that are present in the ground truth, while the recall gives the proportion of ground truth types that were inferred. The F1-score provides an average of precision and recall.

5.2 Experimental Results and Discussion

In this section, we present and discuss the results of our evaluation (cf. Table 2 and 3). We recall edge type refers to the union of ordinary and inheritance edge types (cf. Definition 3.2).

5.2.1 Quality of the Schema Inference. In both the Neuprint and LDBC datasets, the property value data types, as well as the edge cardinality constraints of the correctly identified edge types, have been inferred accurately. The precision, recall and F1-score of the node and edge types (cf. Table 4) demonstrate the overall good quality of the types inferred with our label-oriented approach. We could not compute these metrics for the Covid19 dataset due to the lack of a ground truth on this schema.

Dataset	Nodes	Edges	Node Labels	Edge Labels	Unlabel. Nodes	Nested or Multiple Values
mb6	486,267	961,571	10	3	0	Yes
fib25	802,479	1,625,439	10	3	0	Yes
covid19	10,447,251	25,340,047	60	73	5	Yes
LDBC	1,577,397	8,179,418	7	14	0	No

Table 1: Characteristics of the datasets used in the study.

	Baseline		label-oriented-variant				
Dataset	**Node Types**	**Edge Types**	**Node Types**	**Edges Types**	**Inheritance Edges Types**	**Max Node Hierarchies Depth**	**Overlapping Types**
mb6	10	64	5	10	1	1	No
fib25	10	64	5	10	1	1	No
covid19	60	75	77	159	43	1	Yes
LDBC	7	21	7	21	0	0	No

Table 2: Inferred types with the label-oriented variant.

Dataset	Node Types	Edge Types	Inheritance Edges Types	Max Node Hierarchies Depth	Overlapping Types
mb6	68	795	786	9	Yes
fib25	47	427	418	8	Yes
LDBC	17	72	51	5	Yes

Table 3: Inferred types with the property-oriented variant.

	Label-oriented						Property-oriented					
	Precision		Recall		F1		Precision		Recall		F1	
Dataset	N	E	N	E	N	E	N	E	N	E	N	E
mb6	0.80	0.83	0.80	0.83	0.80	0.83	0.29	0.01	0.80	0.83	0.43	0.01
fib25	0.80	0.83	0.80	0.83	0.80	0.83	0.09	0.01	0.80	0.83	0.15	0.27
ldbc	1.00	1.00	0.54	0.70	0.70	0.82	0.47	0.47	0.62	0.75	0.53	0.58

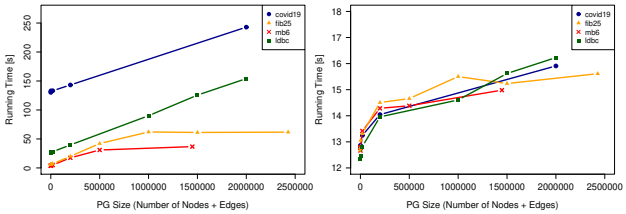
Table 4: Precision, recall and F1 of the inferred types (N is for node types, E is for edge types).

Baseline Comparison. We first compare the schemas inferred by our label-oriented approach with a baseline, the schemas returned by the Neo4j `call db.schema` query (cf. Table 2). The latter outputs many spurious types as it only targets single-labeled node types, even in the presence of multi-labeled node instances. Moreover, no property types or cardinality constraints can be captured, as opposed to our proposed method. As a result, the baseline schema is not accurate and error-prone.

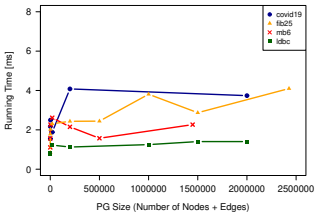
Label-Oriented Approach. All three metrics—notably the precision, with a 0.80 to 1.00 range—are reasonably high. They are identical for both Neuprint datasets (mb6 and fib25), as they share both the ground-truth and inferred types. Only one node type and its corresponding inheritance edge type absent in the ground-truth schema have been mistakenly identified. This is due to an inconsistency in the labeling of this particular node type where some of its instances have more labels than others. Hence, our algorithm incorrectly aggregated them into distinct node types. This highlights the sensitivity of our method to noisy labels. A statistical approach to the type inference, such as clustering methods [9], would allow to group together nodes that share similar, but not identical, label sets. Combining it with graph embedding [15], which maps the input graph to a low-dimension space while preserving the inherent characteristics of the graph to the best possible extent, like in [11], could also emerge as a promising solution. Data types and node hierarchies would still need to be inferred, possibly by integrating such approaches with our schema inference method.

In the LDBC graph, all inferred types exist in the ground-truth schema but none of the ground-truth hierarchies were discovered. Indeed, they were either defined via a `type` property, instead of labels, or identifiable only through properties in common. The former might be addressed with a semantic approach, while the latter is partially overcome with our property-oriented variant.

Property-Oriented Approach. The low precision and F1 scores obtained with the property-oriented approach may stem from the inference of numerous spurious types—in addition to the correct ones. For instance, in the mushroombody dataset (mb6)



(a) Step 1: Preprocessing Queries. (b) Step 2: Types inference.



(c) Step 3: Node hierarchy inference.

Figure 4: Average running times for various graphs.

63 additional node types were inferred. Indeed, since we are considering property sets to infer node types, for a given label set, we infer as many node types as there are combinations of properties—although in some cases, this behavior is expected (cf. Example 4.4.1). Furthermore, different node types may have common property keys even if they are not subtypes of a common supertype. For example, in the LDBC graph, the node types `Place` and `Person` both hold a property with the key `name`.

Nonetheless, in the Neuprint graphs, the number of TPs and FPs remain unchanged from one variant to the other. The recall remains thus constant. Even more remarkably, in the LDBC graph, more types present in the ground-truth are identified than with the label-oriented variant. They correspond to supertypes that could not be inferred using solely the node labels. This is reflected in the recall scores, which increase from 0.54 to 0.62, for nodes, and from 0.70 to 0.75, for edges.

Label-Oriented vs Property-Oriented Approaches. To summarize, the label-oriented variant outputs schemas with a very good precision. However, it is missing node types that can only be inferred through property-related information. This is partially overcome by the property-oriented variant, which is marked by an improved recall. Nonetheless, since many spurious types are inferred as well, this is done at the expense of the precision. Hence, the label-oriented variant should be preferred, either when the node hierarchies in the input PG are defined through labels, or when there are no hierarchies—such as in the Neuprint graphs. On the other hand, the property-oriented variant should be picked when properties are crucial to the inference process, such as in the presence of unlabeled nodes or when hierarchies are determined by property co-occurrence information.

5.2.2 Scalability We obtained the average running times of our schema inference pipeline for portions of different sizes of the datasets. The times discussed in this section were acquired with our label-oriented implementation. Those from our property-oriented implementation are of the same order of magnitude. The first step (cf. Fig. 4a), where we match every single node and edge of the input PG, brings to light the problem of the overhead caused by the Cypher queries, which increases with the size of the input PG. Indeed, the running times can go up to about 1900s for the complete covid19 dataset, with its 10M nodes and 25M edges (this data point is not represented in Fig. 4a to improve legibility). As such, the pipeline running time is dominated by this step. Still, it seems that it is at worst linear in the input size. Fig. 4b displays the sublinear behavior of the running times of Step 2, which is as expected with regards to [2]. Moreover, our parsing function has an average running time smaller than 2ms, which is satisfactory. On average, Step 3 (cf. Fig. 4c) runs in less than 5ms and is constant when the input PG size increases, which comforts our complexity analysis carried out in Section 4.3. Additionally, Step 2 and 3 are not sensitive to the heterogeneity in the complexity of data types and structures displayed in the different datasets.

6 Conclusion and Future Work

We have presented a novel end-to-end schema inference method for PGs that handles complex and nested property values, multi-labeled nodes, node hierarchies overlapping node types, edge cardinality constraints and optionality of properties.

We have proposed and empirically evaluated two variants that both scale well. The *label-oriented* variant provides an inferred schema of good quality. One of its main shortcomings is the loss of property co-occurrence information that could lead to additional supertype identification. This is resolved by our *property-oriented* approach, which concurrently improves recall scores. However, in the process, many extraneous types are inferred. A solution worth exploring in future work would be to find a non-trivial way to combine the outputs of these two variants to exclusively retain the wanted node types.

Our schema inference method remains sensitive to variation in the labels and property keys, be it due to inconsistent or multilingual naming. To overcome this, it would be interesting to consider a clustering step on the nodes and edges of the input instances, possibly combined with graph embeddings taking into account semantic information to simplify graph representations.

References

[1] Renzo Angles. 2018. The Property Graph Database Model. *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management* (2018).

[2] Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *The VLDB Journal* 28, 4 (2019).

[3] Peter W. Battaglia and et al. 2018. Relational inductive biases, deep learning, and graph networks. *ArXiv* abs/1806.01261 (2018).

[4] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. Vol. 10. Morgan & Claypool Publishers. 1–184 pages.

[5] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema Validation and Evolution for Graph Databases. In *Conceptual Modeling*. Springer International Publishing, Cham, 448–456.

[6] Redouane Bouhamoum, Kenza Kellou-Menouer, Zoubida Kedad, and Stéphane Lopes. 2018. Scaling Up Schema Discovery for RDF Datasets. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*. 84–89.

[7] Olaf Hartig and Jan Hidders. 2019. Defining schemas for property graphs by using the GraphQL schema definition language. In *Proceedings of GRADES/NDA Workshop*.

[8] LDBC Social Network Benchmark task force. 2019. *The LDBC Social Network Benchmark (version 0.3.2)*. Technical Report.

[9] Artem Lutov, Soheil Roshankish, Mourad Khayati, and Philippe Cudré-Mauroux. 2018. StaTIX - Statistical Type Inference on Linked Data. In *Proceedings of Big Data*.

[10] Silvio Normey Gómez, Lorena Etcheverry, Adriana Marotta, Silvio Normey, and Mariano P Consens. 2018. Findings from Two Decades of Research on Schema Discovery using a Systematic Literature Review. In *AMW*.

[11] Benedek Rozemberczki, Ryan Davies, Rik Sarkar, and Charles Sutton. 2019. GEMSEC: Graph Embedding with Self Clustering. *ASONAM* (2019), 65–72.

[12] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2-3 (2020), 595–618.

[13] Shin-ya Takemura and et al. 2015. Synaptic circuits and their variations within different columns in the visual system of Drosophila. *Proceedings of the National Academy of Sciences* 112, 44 (Nov 2015), 13711–13716.

[14] Shin-ya Takemura and et al. 2017. A connectome of a learning and memory center in the adult Drosophila brain. *eLife* 6 (Jul 2017).

[15] Q. Wang, Z. Mao, B. Wang, and L. Guo. 2017. Knowledge Graph Embedding: A Survey of Approaches and Applications. *IEEE TKDE* 29, 12 (2017), 2724–2743.

Short Paper

Optimizing SPARQL Queries using Shape Statistics

Kashif Rabbani
Aalborg University, Denmark
kashifrabbani@cs.aau.dk

Matteo Lissandrini
Aalborg University, Denmark
matteo@cs.aau.dk

Katja Hose
Aalborg University, Denmark
khose@cs.aau.dk

ABSTRACT

With the growing popularity of storing data in native RDF, we witness more and more diverse use cases with complex SPARQL queries. As a consequence, query optimization – and in particular cardinality estimation and join ordering – becomes even more crucial. Classical methods exploit global statistics covering the entire RDF graph as a whole, which naturally fails to correctly capture correlations that are very common in RDF datasets, which then leads to erroneous cardinality estimations and suboptimal query execution plans. The alternative of trying to capture correlations in a fine-granular manner, on the other hand, results in very costly preprocessing steps to create these statistics. Hence, in this paper we propose *shapes statistics*, which extend the recent SHACL standard with statistic information to capture the correlation between classes and properties. Our extensive experiments on synthetic and real data show that shapes statistics can be generated and managed with only little overhead without disadvantages in query runtime while leading to noticeable improvements in cardinality estimation.

1 INTRODUCTION

Driven by diverse movements, such as Linked Open Government Data, Open Street Map, DBpedia [3], and YAGO [21], more and more data is being published in RDF [7] capturing a multitude of diverse information. Along with the growing popularity, increasingly complex queries formulated in SPARQL [6] are being executed over such data to answer business and research questions. Query logs of the public DBpedia SPARQL endpoint, for instance, contain SPARQL queries with up to 10 joins [4] and analytic queries in the biomedical field can involve more than 50 joins per query [9]. Therefore, the need for high-performance SPARQL query processing is now more pressing than ever.

Existing approaches for query optimization in RDF stores often adapt techniques from relational databases modeling an RDF dataset as a single large table with three column [5, 16] (one column for each of the components of an RDF triple: subject, predicate, and object). Nevertheless, accurate cardinality estimation is at the heart of any query optimizer that does not rely on heuristics but instead uses a cost model to find the best query execution plan for a given query. Cardinality estimation then relies on the availability of statistics describing the characteristics of the data to estimate the sizes of intermediate results produced while query execution. However, general statistics typically result in highly imprecise estimations since they are mostly gathered on the RDF graph as a whole, in contrast to the relational case where it is possible to create such statistics with higher precision since data is separated into multiple tables [15]. Furthermore, assuming independence when joining parts of SPARQL queries (triple patterns) leads to erroneous estimations [9] as co-occurrences of certain predicates are highly correlated [19].

Hence, exploiting more fine-grained statistics capturing correlations among RDF triples leads to more accurate join cardinality estimations [19]. However, creating such statistics comes at the price of a very time and resource-intensive preprocessing step. On the other hand, the alternative of online, query-dependent, sampling [20] results in overheads during query optimization. Instead, what we propose in this paper is to better exploit the information that is often provided along with an RDF dataset: SHACL (Shapes Constraint Language) [14] constraints, which is a recent standard for validating RDF datasets that are becoming more and more popular. SHACL defines so-called shapes describing the relationships between entities of a specific class, their properties, and their connections to other classes of entities. Although they are currently only used for validation purposes, we show in this paper that by slightly extending them with basic statistics, they can also be exploited for join cardinality estimation.

In summary, this paper makes the following contributions. First, we extend the SHACL definition to capture statistical information to replace the need for creating complex (and expensive) statistics over RDF datasets. To the best of our knowledge, this is the first proposal of this kind. Second, we introduce an algorithm to enhance SHACL shapes with statistical information and to exploit these statistics for join cardinality estimation and query optimization. Third, we study the impact of our approach using both synthetic (LUBM [10], WatDiv [2]) and real (YAGO-4 [21]) datasets, demonstrating that shapes statistics can provide higher precision for query optimization with only a little overhead.

This paper is structured as follows. While Sections 2 and 3 discuss related work and introduce preliminaries, Section 4 formally defines the problem. Section 5 then describes our proposed extension of the SHACL standards, and Section 6 presents techniques to exploit the additional information for cardinality estimation and query optimization. Section 7 discusses the results of our extensive experimental study, and Section 8 concludes the paper with an outlook to future work.

2 RELATED WORK

Cardinality estimation has been studied extensively in the context of relational databases [20]. For SPARQL queries, existing techniques adapt relational approaches [13, 24] and focus mostly on specific type of queries [19]. Usually, these approaches construct different kinds of single or multidimensional synopses over databases that can be used to estimate cardinalities [23]. While algorithms designed to generate synopsis for unlabelled graphs are not applicable here (as the edges in RDF graphs are labeled), consequently approaches to generate RDF summaries either produce very large summaries [23], have very high computational complexities, or they are unable to preserve the RDF schema while constructing the summaries [23]. Therefore, the most promising approaches aim at using statistics computed directly from edge label frequencies. In particular, RDF-3X proposes a histogram-based technique for cardinality estimation based on edge label frequencies. This technique was later extended by exploiting the statistical information of Characteristic Sets [19], which compute frequencies of sets of predicates sharing the same

subject to estimate the cardinalities. This approach shows high performance for star-shaped queries while it suffers from significant underestimation due to the independence assumption in the general case [20]. This approach was extended as Characteristic Pairs [18] to overcome this limitation, but it could only support multi-chain star queries. Moreover, extracting Characteristic Sets from large heterogeneous graphs is computationally expensive. SumRDF [23] is another cardinality estimation approach based on a graph summarization. It fails to handle large queries due to a prohibitive computation cost, and it is costly to construct such summaries over large RDF graphs [20].

A recent benchmark, G-CARE [20], analyzed the performance of existing cardinality estimation techniques for subgraph matching. This analysis revealed that the techniques based on sampling and designed for online aggregation outperform the cardinality estimation techniques for RDF graphs. *This calls for a more in-depth study on how to perform cardinality estimation for SPARQL query optimization appropriately.*

In a recent work, Shape Expressions (ShEx) [22] have been used to reorder triple patterns to enable SPARQL query optimization [1], i.e., it estimates an order of execution for the triple patterns based on some heuristic inference on which triples are more selective. For instance, if a shape definition says that every instructor has one or more courses, but every course has exactly one instructor, it infers that the cardinality of courses is at least the same as the cardinality of instructors and probably larger. Hence, this optimization procedure is not based on actual data.

Therefore, contrary to existing works, we aim at exploiting fine-grained statistics based on shapes to produce more precise cardinality estimations for query planning. This will allow us to overcome the limitations of existing methods that only use the global-statistics [11]. To this end, instead of creating large expensive summaries and characteristic sets over the RDF graphs to estimate the cardinalities, we exploit SHACL shapes constraints (which are as expressive as ShEx [22]) and annotate the *Node* and *Property Shapes* with the statistics of the input RDF graph. Compared to other solutions, it requires a lightweight preprocessing and retains the structure of original RDF and SHACL shapes graphs. Moreover, *this allow us to study more closely the effect of more fine-grained statistics, and more accurate cardinality estimation for the task of SPARQL query optimization.*

3 PRELIMINARIES

RDF Graphs: RDF graphs model entities and their relationships in the form of triples consisting of SPO <*subjects, predicates, objects*>. We present a simplified example of an RDF graph G based on the LUBM [10] dataset in Figure 1, where oval and rectangular shapes represent IRIs and literal nodes, respectively. An RDF graph is formally defined as:

Definition 3.1 (RDF Graph). Given pairwise disjoint sets of IRIs I, blank nodes B, and literals L, an RDF Graph G is a finite set of RDF triples $\langle s, p, o\rangle \in (I \cup B) \times I \times (I \cup B \cup L)$.

SPARQL: SPARQL [6] is a standard query language for RDF. A SPARQL query consists of a finite set of triple patterns (known as basic graph pattern, BGP) and some conditions that have to be met in order for data to be selected and returned from an RDF graph. Each SPO position in a triple pattern can be concrete (i.e., bound) or a variable (i.e., unbound). The variable names in a SPARQL query are prefixed by a '?' symbol, e.g., ?X. To answer a BGP, we require *a mapping between variables to values in an RDF graph*, all the resulting triples existing in the RDF graph

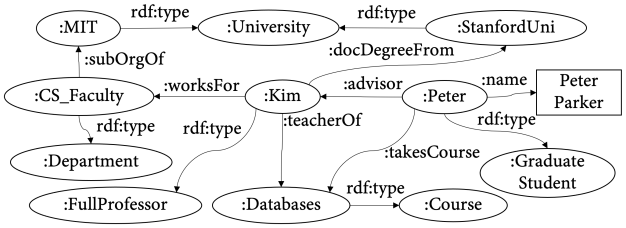


Figure 1: An RDF Graph G

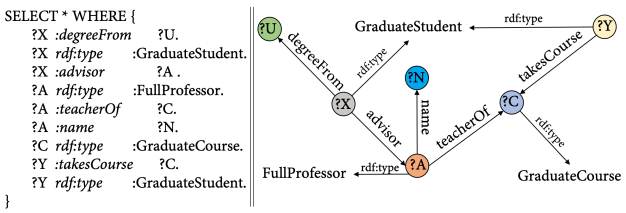


Figure 2: Query Q and its Graph Q_G

obtained by replacing the variables with values are answers to the BGP. Figure 2 shows an example SPARQL query (Q) and its query graph Q_G on the graph of Figure 1. A BGP is defined as:

Definition 3.2 (BGP). Given a set of IRIs I, literals L, and variables V, a BGP is defined as $T \subseteq (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$, whose elements are called triple patterns.

Shapes Graphs: Several schema languages have been proposed for RDF in the past, where the most common are RDF Schema (RDFS[1]) and OWL [17]. RDFS is primarily used to infer implicit facts, and OWL is an extension of RDF and RDFS to represent ontologies. The declarative Shapes Constraint Language (SHACL) [14] became a W3C standard recently. SHACL schema provides high-level information about the structure and contents of an RDF graph. It allows to define and validate structural constraints over RDF graphs. SHACL models the data in two components: the *data graph* and the *shape graph*. The *data graph* contains the actual data to be validated, while the *shape graph* contains the constraints against which resources in the *data graph* are validated. These constraints are modeled as node and property shapes, which consist of attributes encoding the constraints. The node shapes constraints are applicable on nodes that are instances of a specific type in the *data graph* while the property shapes constraints are applicable to predicates associated with nodes of specific types. We define a SHACL shapes graph as follows:

Definition 3.3 (SHACL Shapes Graph). A SHACL shapes graph G_{sh} is an RDF graph describing a set of node shapes S and a set of property shapes P, such that $target_S : S \mapsto I$ and $target_P : P \mapsto I$ are injective functions mapping each node shape $s_i \in S$ and each property shape $p_i \in P$ to the IRI of a target class and a target predicate in G respectively, and $\phi : S \mapsto 2^P$ is a surjective function assigning to each node shape s_i a subset $P_i \subseteq P$ of property shapes.

For example in Figure 3, node shape constraints are applicable on node `ub:GraduateStudent` and its property shapes constraints are applicable on predicates like `takesCourse`, and `advisor`. This information is declared with attributes *sh: targetClass* for node shapes and *sh:path* for property shapes. Note that the attributes in the dark shaded boxes are part of our extension of the SHACL definition, explained in Section 5.

[1] https://www.w3.org/TR/rdf-schema/

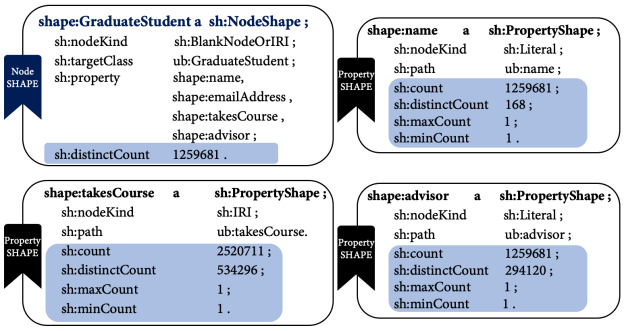


Figure 3: SHACL Shapes Graph

The Shapes Expression (ShEx [22]) language also serves a similar purpose as SHACL to validate RDF graphs. Nonetheless, the two formulations diverge mostly at the syntactic level [12], and our approach can be extended to work using ShEx or other constraints languages as well without the loss of generality.

4 PROBLEM FORMULATION

Given an input query Q, a query optimizer has the goal to find a query plan expected to answer Q in the minimum amount of time [15]. Constructing a SPARQL query plan includes finding a join ordering between triple patterns of its BGPs. In this paper, we focus on the join ordering of BGPs defined as follows:

Definition 4.1 (Join Ordering). Given a set of triple patterns $T=\{tp_1, tp_2, ..., tp_n\} \subseteq (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$, the join order O for BGPs is defined as a total ordering O of T so that for every $t_i, t_j \in T$ either $t_i \prec_O t_j$ or $t_j \prec_O t_i$.

To find an optimal plan, a query optimizer needs to explore the search space of semantically equivalent join orders and choose the optimal (cheapest) plan according to some cost function. It is crucial to accurately estimate the join cardinality between triple patterns of a given query to construct a query plan with an efficient join ordering [9]. In line with the related work [20], we neglect other cost factors and focus on join cardinality as the most dominant cost factor to find a join ordering. We formally define the problem of estimating join cardinalities as follows:

Problem 1 (Join Cardinality Estimation). *Given a set of triple patterns $T=\{tp_1, tp_2, ..., tp_n\}$, apply a cardinality estimation function $\bar{J} : T \times T \mapsto \mathbb{N}$ such that for every pair of triple patterns $(tp_i, tp_j) \in T$, $\bar{J}(tp_i, tp_j) \approx |tp_i \bowtie tp_j|$.*

We extend the above estimation problem also to the case of joining a triple pattern with the intermediate results of prior join operations, e.g., to estimate the total cardinality $\bar{J}((tp_i \bowtie tp_j), tp_k) \approx |(tp_i \bowtie tp_j) \bowtie tp_k|$. Then, given such estimates, an optimal query plan minimizes the total number of operations to compute, i.e., the execution costs $Cost(T, O)$ of the order O for the set T. In practice, this total join cost is obtained by summing up the intermediate cardinalities of each join operation in their respective join order. Hence, we formalize the problem of join order optimization as follows:

Problem 2 (Join Order Optimization). *Given a set of triple patterns $T=\{tp_1, tp_2, ..., tp_n\}$ and a join cardinality estimation function $\bar{J}$, find the join order O obtained as* $\arg\min_O Cost(T, O)$.

5 EXTENDING SHACL WITH STATISTICS

To compute more accurate join cardinality estimations (Problem 1), we capture the correlations between RDF triples by extending SHACL's node and property shapes with fine-grained statistics of the RDF graph. We denote these statistics as *shapes statistics*. These include the total triple count (*sh:count*), minimum (*sh:minCount*) and maximum (*sh:maxCount*) number of triples for each instance, and the number of distinct objects for property instantiations (*sh:distinctCount*). The attributes shown in the dark shaded boxes in Figure 3 are the annotated statistical attributes of their respective node and property shapes. These statistics are computed by executing analytical SPARQL queries over the RDF graph. For instance, to compute the number of instances of *GraduateStudent* in the dataset, i.e, the value of attribute *sh:count* of node shape *GraduateStudent*, the annotator issues the SPARQL query: `SELECT COUNT(*) WHERE {?x a ub:GraduateStudent}`.

Along with *shapes statistics*, we also define *global statistics* by extending VOID[2] statistics with more precise statistics of RDF properties, i.e., the distinct subject count (DSC) and distinct object count (DOC) of each property of the RDF graph.

6 QUERY PLANNING

In this section, we present our approach to exploit global and shapes statistics to obtain more accurate join cardinality estimates (Problem 1). These estimates, in turn, are used for join order optimization (Problem 2).

6.1 Cardinality Estimation of Triple Patterns

A SPARQL query contains joins between multiple triple patterns. Hence, the first step is to estimate how many triples match every triple pattern individually. We exploit the statistical information contained in the extended SHACL shapes graph (Section 5) to obtain this estimate. Hence, for each triple pattern, we obtain their corresponding node or property shapes using the values of the *sh:targetClass* and *sh:path* attributes.

First, all triples of the type *<?x, a, [Class]>* (i.e., instances with *rdf:type [Class]*) are mapped to the node shape having that class as the value of the attribute *sh:targetClass*. Then, triples having variable *?x* as a subject are also assigned to that node shape. The triple predicate determines instead its corresponding candidate property shapes, i.e., those with a matching value for *sh:path*. For example, given triples tp_1=*<?x, rdf:type, ub:GraduateStudent>* and tp_2=*<?x, ub:name, ?n>*, the subject *?x* is assigned to node *shape:GraduateStudent*, while the predicate in tp_2 matches *shape:name* (Figure 3, top left and top right).

Once the candidate shapes for all the triple patterns are identified, their statistical information combined with the distinct subject and object count (DSC & DOC) from the *global statistics* are used in combination with the formulas shown in Table 1 to compute their expected cardinality. These formulas, inspired by a previous work [11], cover all possible types of triple patterns. The term c_X in the formulas denotes the count of X in the RDF graph; $c_{triples}$ denotes the count of all triples and $c_{objects}$ the count of all objects. Similarly, c_{X_Y} represents the count of X having Y. This can be used, for instance, to derive that there are ~ 85K triples matching *<?x, rdf:type, ub:FullProfessor>* (Table 2a). While both global and shapes statistics can be used to estimate the cardinality of triple patterns using these formulas, they can lead to different estimated cardinalities. When the query does not contain any type-defined triple, only global statistics are used.

[2]Vocabulary of Interlinked Datasets: https://www.w3.org/TR/void/

Triple Pattern	Cardinality	Triple Pattern	Cardinality
?s ?p obj	$\frac{c_{triples}}{c_{objects}}$	?s ?p ?o	$c_{triples}$
subj ?p obj	$\frac{c_{triples}}{c_{distSubj} \times c_{distObj}}$	subj ?p ?o	$\frac{c_{triples}}{c_{distSubj}}$
?s pred obj	$\frac{c_{pred}}{c_{pred_{obj}}}$	?s pred ?o	c_{pred}
subj pred obj	$\frac{c_{pred}}{c_{distSubj} \times c_{distObj}}$	sub pred ?o	$\frac{c_{pred}}{c_{pred_{sub}}}$
?s rdf:type obj	$c_{entities_{rdf:type_{obj}}}$	?s rdf:type ?o	$c_{rdf:type}$
subj rdf:type obj	1 *or* 0	subj rdf:type ?o	$\frac{c_{rdf:type}}{c_{rdf:type_{sub}}}$

Table 1: Cardinality estimation of triple patterns

6.2 Cardinality Estimation of Joins

The join operation is performed on a common variable between two triple patterns. We consider three possible types of joins between two triple patterns based on the position of the common variable, namely: Subject-Subject (SS), Subject-Object (SO), and Object-Object (OO). If there is no common variable between two triple patterns, the join will result in a Cartesian product. Inspired by related work [8], we estimate the SS, SO, and OO join cardinalities using the formulas stated in Equations 1, 2, and 3. Note that DSC_i and DOC_i in the formulas represent the distinct subject and object count of triple pattern i respectively.

$$\widehat{card}\,(tp_i \bowtie_{SS} tp_j) = \frac{card_i \times card_j}{max\,(DSC_i, DSC_j)} \quad (1)$$

$$\widehat{card}\,(tp_i \bowtie_{SO} tp_j) = \frac{card_i \times card_j}{max\,(DSC_i, DOC_j)} \quad (2)$$

$$\widehat{card}\,(tp_i \bowtie_{OO} tp_j) = \frac{card_i \times card_j}{max\,(DOC_i, DOC_j)} \quad (3)$$

6.3 Join Ordering

Given an RDF graph G, its shapes statistics graph (G_{sh}), and global statistics graph (G_{gs}), we propose an algorithm to compute the join ordering for an input query Q (Algorithm 1). In the first step, the triple patterns of Q are sorted in ascending order of their estimated cardinalities using only global statistics. The algorithm starts with the triple pattern having the least cardinality and then estimates its join cardinality with the rest of the triple patterns using the formulas from Section 6.2. The algorithm iterates over all the triple patterns and chooses a triple pattern with the least estimated join cardinality (size of intermediate result) given the triple already selected. This produces a first join ordering based on global statistics. In the second step, shapes statistics are taken into account, and both the estimated cardinalities and the join ordering proposed in the first step are revised using these shapes specific fine-grained statistics. The algorithm also computes the cost of each join ordering by adding the estimated join cardinalities in each iteration. Its complexity is cubic to the number of triple patterns in the query, i.e., $O(n^3)$.

Given our example query Q, and the cardinalities of its triple patterns T=$\{tp_1, tp_2, ..., tp_9\}$ estimated with both global and shape statistics, Tables 2a and 2b show the join ordering computed only using global statistics (O_{gs}) and via shapes statistics (O_{ss}), respectively. There is a significant difference between the estimated and true join cardinalities and their final total cost. The estimated join cardinalities for O_{ss} are much closer to the true cardinalities of the query than the estimates for O_{gs} with two exceptions for tp_5 and tp_8 where shapes statistics largely overestimate their cardinalities due to skewed distribution of data.

Algorithm 1 Join Ordering

Input: Q, G, G_{sh}, G_{gs}
Output: Join order O of Q
$p \leftarrow [\,]$; $r \leftarrow [\,]$; ▷ p: processed, r: remaining
$cost \leftarrow 0$; $card \leftarrow 0$; $queue \leftarrow queue.init()$;
$tps \leftarrow getTPs(Q)$;
$tps_{\triangle} \leftarrow getCandidateShapes(Q, G, G_{sh}, G_{gs})$;
$tps' \leftarrow computeCardinalities(tps_{\triangle})$; ▷ Table 1
$sort(asc, tps'.cardinality)$;
$p.add(tps'_0)$; $r.addAll(tps' - tps'_0)$;
$cost = tps'_0.cardinality$;
$queue.add(tps'_0.index)$;
for $tp_i \in tps'$ **do** ▷ i > 0
  $index = tp_i.index$; $cost_{local} = cost$;
  $queue' = queue$;
  while $!queue'.isEmpty$ **do**
    $tp_a = queue'.poll()$;
    for $tp_b \in r$ **do**
      $c = 0$;
      if $tp_a \bowtie_T tp_b$ **then** ▷ $T \in \{SS, SO, OS, OO\}$
        $c = \bar{J}(tp_a, tp_b)$; ▷ $\bar{J} : T \times T \mapsto \mathbb{N}$ (Prob 1)
      else $c = cp(tp_a, tp_b)$; ▷ Cartesian Product
      if $c < cost_{local}$ **then**
        $cost_{local} = c$; $index = tp_b.index$; $card = c$;
    $cost$+=$cost_{local}$;
  $queue.add(index)$; $p.add(tps'.get(index))$;
  $r.remove(tps'.get(index))$;
$O \leftarrow queue.poll()$;

7 EXPERIMENTAL EVALUATION

We investigated the performance of query plans proposed using our algorithm (with global and shapes statistics) compared to the plans proposed by two state-of-the-art query engines (Apache Jena ARQ[3] and GraphDB[4]) as well as two state-of-the-art RDF cardinality estimation approaches (Characteristic Sets [19] and SumRDF [23]). All experiments are performed on a single machine with Ubuntu 18.04, having 16 cores and 256GB RAM.

Datasets: We used LUBM [10], WatDiv [2], and YAGO-4 [21] to study various query plans on different datasets and sizes (Table 3). In particular, we used LUBM-500, two variants of WatDiv datasets (WATDIV-S (Small) with ~108.9 M triples and WATDIV-L (Large) with 1 billion triples), and for YAGO-4 we used the subset containing instances that have an English Wikipedia article.

Implementation: Nowadays, constraints languages are having widespread application to validate RDF graphs [21]. We assume the availability of SHACL shapes graph with the dataset and provide a *Shapes Annotator* to extend it with statistics of the graph. For cases where they are not present, the SHACLGEN[5] library is commonly used to generate shapes graphs and we also use it in our case (e.g., for YAGO-4). All shapes are then extended with the required statistics using our *Shapes Annotator* (implemented in Java). The SHACL shapes graph for LUBM, for instance, is 45 KB, and the size of extended shapes is 68 KB. The time required to extend the SHACL shapes depends on the number of its nodes and property shapes. The process of extending LUBM shapes graph took 16 minutes, WATDIV-S took 8 minutes, and for YAGO-4 (which consists of 8888 nodes and 80831 property shapes) it took 62 minutes. We implemented our join ordering algorithm in Java using Jena[3]. The source code is available on our website[6].

We loaded all three datasets and their relevant SHACL shapes graphs into Jena TDBs[3]. We used our join ordering algorithm to construct query plans using global and shapes statistics. For Jena, we used its ARQ query engine to obtain the query plans. For GraphDB, we loaded all datasets in GraphDB and used its *onto:explain* feature to obtain the query plans. For Characteristic Sets [19] approach, we generated characteristic sets of each

[3]https://jena.apache.org/documentation/
[4]https://graphdb.ontotext.com
[5]https://pypi.org/project/shaclgen/
[6]https://relweb.cs.aau.dk/rdfshapes/

	Triple Pattern (TP)	DSC	DOC	E_{TP} Card	$E_{\bowtie}$ Card	$T_{\bowtie}$ Card
1:	?A *rdf:type* :FullProfessor	85,006	85,006	85,006		
2:	?A :*name* ?N	10,696,541	1,480	10,696,541	85,006	85,006
3:	?A :*teacherOf* ?C	359,795	1,079,580	1,079,580	8,579	255,148
4:	?C :*advisor* ?A	2,052,228	299,177	2,052,228	1,646	2,055,430
5:	?X *rdf:type* :GraduateCourse	539,467	539,467	539,467	822	1,027,909
6:	?X *rdf:type* :GraduateStudent	1,259,681	1,259,681	1,259,681	504	630,419
7:	?X :*degreeFrom* ?U	1,619,476	1,000	2,337,985	575	630,419
8:	?Y :*takesCourse* ?C	5,220,814	1,074,409	14,405,077	7,674	2,964,894
9:	?Y *rdf:type* :GraduateStudent	1,259,681	1,259,681	1,259,681	1,851	2,964,894
					∑ =106,657	∑ =10,614,119

(a) Join ordering using Global Statistics (O_{gs})

	Triple Pattern (TP)	DSC	DOC	E_{TP} Card	$E_{\bowtie}$ Card	$T_{\bowtie}$ Card
1:	?A *rdf:type* :FullProfessor	85,006	85,006	85,006		
2:	?A :*name* ?N	85,006	10	85,006	85,006	85,006
3:	?A :*teacherOf* ?C	85,006	255148	255,148	85,006	255,148
4:	?C *rdf:type* :GraduateCourse	539,467	539,467	539,467	255,148	255,148
5:	?X :*advisor* ?A	2,052,228	299,177	2,052,228	1,750,207	127,523
6:	?X *rdf:type* :GraduateStudent	1,259,681	1,259,681	1,259,681	1,074,297	1,027,909
7:	?X :*degreeFrom* ?U	1,259,681	1,000	1,259,681	659,416	630,419
8:	?Y :*takesCourse* ?C	5,220,814	1,074,409	5,220,814	8,841,082	2,964,894
9:	?Y *rdf:type* :GraduateStudent	1,259,681	1,259,681	1,259,681	2,133,181	2,964,894
					∑ =14,883,343	∑ =8,310,941

(b) Join ordering using Shapes Statistics (O_{ss})

Table 2: This table shows the statistics (distinct subject count (DSC) and distinct object count (DOC)) of each triple pattern, the estimated cardinality of each triple pattern (E_{TP}), the estimated join cardinality ($E_{\bowtie}$ Card) and the true join cardinality ($T_{\bowtie}$ Card) for the ordered triple patterns of example query Q computed over LUBM dataset.

	LUBM	WATDIV-S	WATDIV-L	YAGO-4
# of triples	91 M	108 M	1,092 M	210 M
# of distinct objects	12 M	9 M	92 M	126 M
# of distinct subjects	10 M	5 M	52 M	5 M
# of distinct RDF type triples	1 M	25 M	13 M	17 M
# of distinct RDF type objects	39	46	39	8,912

Table 3: Size and characteristic of the datasets

dataset and used Extended Characteristic Sets [18] to optimize query plans for non-star type queries. Generating characteristic sets for large RDF graphs is computationally expensive. For instance, it took 6.2 hours to generate Characteristic Sets for LUBM, 1.2 hours for WATDIV-S, and 8.2 hours for YAGO-4.

For SumRDF [23], we generated the summaries of each dataset and adapted our join ordering algorithm to exploit their estimates. Similar to Characteristic Sets, the generated summaries require a few GBs of memory and their generation time depends on the size and heterogeneity of the dataset, e.g., it took 4.5 minutes to generate the summary for the LUBM, 14 minutes for WATDIV-S, and 4.3 hours for YAGO-4. We use the same size of LUBM and WatDiv datasets as used in SumRDF [23]. Hence, we used the same parameters to generate their summaries. It is suggested that a reasonable default size for the target SumRDF's summary should be in the order of tens of thousands [23]. Therefore, for YAGO-4, to generate the summary in a reasonable amount of time, we chose 100K as the target size of the summary.

All query plans obtained using these approaches are executed 10x in Jena TDB and each query is interrupted after a timeout of 10 minutes. Since for some approaches the order in which triples are stated in the query matters we shuffle the triple patterns in the BGPs randomly in each iteration before proceeding with query optimization. As the query planning time is always less than 20 milliseconds for all approaches and queries, in the following we focus on analyzing the precision of the cardinality estimation and the resulting query performance.

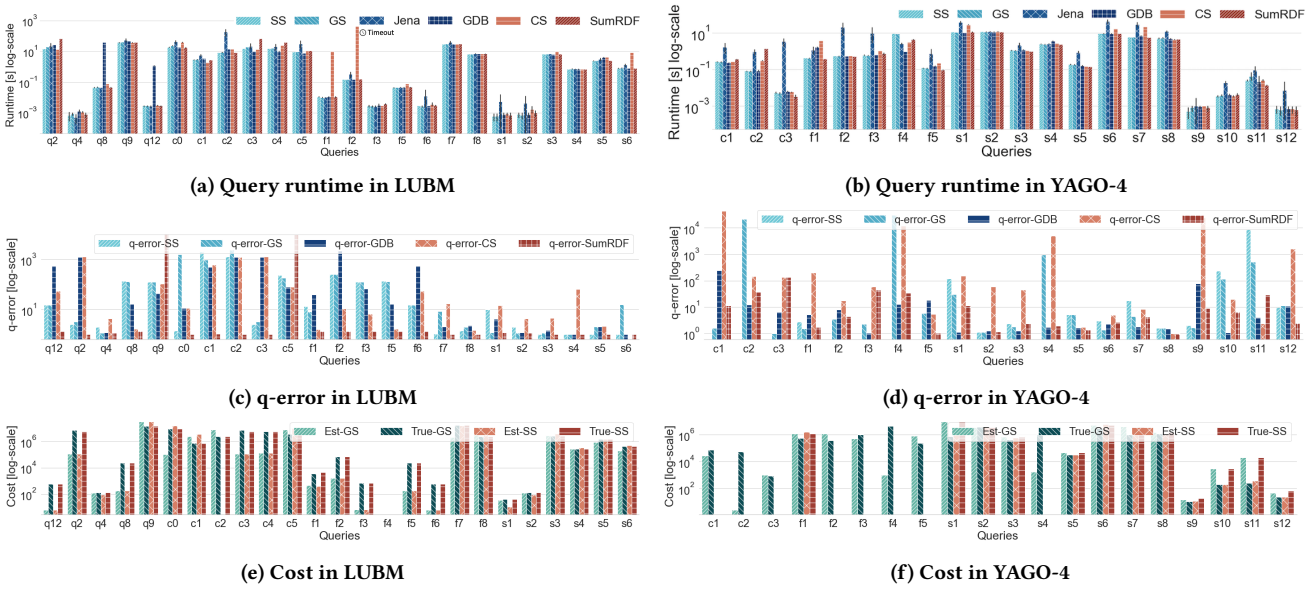
Queries: We distinguish complex (C), snowflake (F), and star (S) queries. LUBM provides 14 default queries that have relatively simple structures. Therefore, we selected queries Q2, Q4, Q8, Q9, Q12 and then created a few additional queries for each category C, F, and S. The WatDiv benchmark includes 3 C, 7 S, and 5 F queries. For YAGO-4 there are no available standard queries or query logs available for benchmarking. Therefore, we have handcrafted 13 queries following the C, F, and S graph patterns from the WatDiv Benchmark. These queries are available on our website[6].

Query Runtime: Due to space constraints, here we only report our findings on LUBM and YAGO-4, results on WatDiv datasets are discussed in the appendix of the extended version[6]. These experiments offer analogous insights to those obtained from the other datasets. Figure 4a shows the query runtime analysis for query plans proposed using the SS approach *(plans constructed by our join ordering algorithm using shapes statistics)*, GS approach *(plans constructed by our join ordering algorithm using global statistics)*, Jena, GraphDB (GDB), Characteristic Sets (CS), and SumRDF on LUBM queries. The query runtime shows that: (i) the plans proposed by the SS approach are more efficient than those obtained with GS for queries having at least one type-defined triple pattern, (ii) the plans proposed by the GS approach are competitive in comparison to the plans of GDB, CS, and SumRDF, (iii) the CS approach is not well suited for large snowflake queries (e.g., F1, F2 (timeout), & F5), and (iv) the plans proposed by Jena are often suboptimal and non-deterministic (shown in the size of the error bars) as it is based on a heuristics-based query optimizer that takes into account the given order of triple patterns in the input query.

Similarly, Figure 4b shows the query runtime for queries on YAGO-4. The query runtime for complex queries (C1, C2, C3) using SS and GS are competitive to the plans proposed by GDB, CS, and SumRDF. Snowflake queries provide interesting insights where each approach behaves differently for every single query. For instance, CS could not find the optimal query plan for queries F1, F3, F4, F5, and SS and GS could not find the most efficient query plan for query F4 due to underestimation of the join cardinalities. However, GDB and SumRDF found almost optimal query plans for all snowflake queries except F1 (GraphDB) and F4 (SumRDF). For star queries, almost all approaches identify plans with comparable good performances. Similar to LUBM, the plans proposed by Jena are rarely the most efficient.

In addition to query runtime, we also report the q-error, which is used to measure the precision of the final query result cardinality estimates [19]. It quantifies the ratio between the estimated ($\hat{c}$) and true result cardinality (c) and is computed as the ratio $max(max(1,c)/max(1,\hat{c}),\ max(1,\hat{c})/max(1,c))$. Ideally, the lower the value of the q-error, the better the estimates are. We analyze the q-error values for SS, GS, GDB, CS, and SumRDF. Figure 4c shows the q-error analysis for LUBM queries. For SS, 15 queries have q-errors lower than 15, 8 queries have q-errors lowever than 250, and only 3 queries have q-errors greater than 250. For GS, 14 queries have q-errors lowever than 15, 8 queries have q-errors lower than 250, and only 4 queries have q-errors greater than 250. Overall, the q-errors for GS and SS are competitive to GDB and CS with few exceptions. However, overall the q-error is very low for SumRDF except queries Q9 and C5. Figure 4d shows the q-error analysis for YAGO-4. For GS and SS, 14 queries have q-errors lower than 15, 2 queries have q-errors lowever than 250, and only 4 queries have q-errors greater than 250. Similar to LUBM, the q-errors of GS and SS are competitive with GDB, CS, and SumRDF with few exceptions.

Finally, Figure 4e and 4f present the analysis between actual and true costs of query plans produced by SS and GS on the LUBM and YAGO-4 datasets. For LUBM, the cost estimated by SS is closer to the actual cost for Q4, Q9, C0, C1, C5, F7, F8, and all



(a) Query runtime in LUBM

(b) Query runtime in YAGO-4

(c) q-error in LUBM

(d) q-error in YAGO-4

(e) Cost in LUBM

(f) Cost in YAGO-4

Figure 4: Query runtime, q-error, and cost analysis on LUBM and YAGO-4

star queries. However, for YAGO-4, the cost estimated by SS is closer to the true cost for almost all queries except C2, F4, and S4. **Summary:** Our results showed that, with only a few exceptions, the query plans proposed using SS and GS are competitive with the other tested approaches on both the synthetic and real data. Overall, the results revealed that our approach is efficient for all examined types of SPARQL queries while requiring only very little overhead to extend SHACL graphs with statistics, which is more efficient and feasible than generating extensive summaries or Characteristic Sets. On average, our approach finds the best query plans for 75% cases on both datasets. For the remaining cases, our approach proposes query plans having an overhead from 14% to 30% on average query runtime w.r.t. the best query plan. Our approach requires 2-4x less preprocessing time, this implies 2 to 6 hours less preprocessing time in our experiments, and 2 orders of magnitude less space.

8 CONCLUSION AND FUTURE WORK

In this paper, we have presented an alternative approach to cardinality estimation for SPARQL query optimization. In particular, we have proposed novel light-weight statistics to capture the correlation in RDF graphs, a cardinality estimation approach, and a join ordering algorithm. We have performed extensive experiments on synthetic and real data to show our approach's effectiveness against two SPARQL query engines and two state-of-the-art RDF cardinality estimators. The results revealed that our approach is efficient in terms of both the preprocessing steps to generate statistics and the cardinality estimation to optimize query plans. Going forward, we plan to integrate our approach with one of the state-of-the-art query engines and enable the support of additional SPARQL query operators.

Acknowledgments. This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B, the EU's H2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 838216, and the Poul Due Jensen Foundation.

REFERENCES

[1] A. Abbas, P. Genevès, Cécile Roisin, and N. Layaïda. 2018. Selectivity Estimation for SPARQL Triple Patterns with Shape Expressions. In *ICWE*. 195–209.
[2] G. Aluç, O. Hartig, M Tamer Özsu, and K. Daudjee. 2014. Diversified stress testing of RDF data management.. In *ISWC*. 197–212.
[3] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. 2009. DBpedia-A crystallization point for the Web of Data. *Journal of web semantics* 7, 3 (2009), 154–165.
[4] A. Bonifati, W. Martens, and T. Timm. 2020. An analytical study of large SPARQL query logs. *The VLDB Journal* 29, 2 (2020), 655–679.
[5] J. Broekstra, A. Kampman, and F. Harmelen. 2002. Sesame: A generic architecture for storing and querying rdf and rdfs. In *ISWC*. Springer, 54–68.
[6] WWW Consortium. 2013. SPARQL 1.1. https://w3.org/TR/sparql11-overview/
[7] WWW Consortium. 2014. RDF 1.1. (2014). https://w3.org/RDF/
[8] H. Garcia-Molina, J. D Ullman, and J. Widom. 2000. *Database system implementation*. Vol. 672. Prentice Hall Upper Saddle River, NJ.
[9] A. Gubichev and T. Neumann. 2014. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*. 439–450.
[10] Y. Guo, Z. Pan, and J. Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics* 3 (2005), 158–182.
[11] S. Hagedorn, K. Hose, K. Sattler, and J. Umbrich. 2014. Resource Planning for SPARQL Query Execution on Data Sharing Platforms. In *International Workshop (COLD) co-located with the 13th ISWC)*, Vol. 1264.
[12] A. Hogan. 2020. Shape Constraints and Expressions. In *The Web of Data*. Springer, Cham, 449–513.
[13] H. Huang and C. Liu. 2011. Estimating selectivity for joined RDF triple patterns. In *CIKM*. ACM, Glasgow, United Kingdom, 1435–1444.
[14] H. Knublauch and D. Kontokostas. 2017. Shapes constraint language (SHACL). *W3C Candidate Recommendation* 11, 8 (2017).
[15] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.
[16] B. McBride. 2002. Jena: A semantic web toolkit. *IEEE* 6, 6 (2002), 55–59.
[17] D. L McGuinness, F. Van Harmelen, et al. 2004. OWL web ontology language overview. *W3C recommendation* (2004).
[18] M. Meimaris, G. Papastefanatos, N. Mamoulis, and I. Anagnostopoulos. 2017. Extended characteristic sets: graph indexing for SPARQL query optimization. In *ICDE*. IEEE, 497–508.
[19] T. Neumann and G. Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*. IEEE, 984–994.
[20] Y. Park, S. Ko, S. S Bhowmick, K. Kim, K. Hong, and Wook-Shin Han. 2020. G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching. In *ACM SIGMOD*. 1099–1114.
[21] T. Pellissier Tanon, G. Weikum, F. Suchanek, et al. 2020. Yago 4: A reason-able knowledge base. In *ESWC*. 583–596.
[22] E Prud'hommeaux, J. Emilio L. Gayo, and H. Solbrig. 2014. Shape expressions: an RDF validation and transformation language. In *ICSS*. 32–40.
[23] G. Stefanoni, B. Motik, and E. V Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *WWW*. ACM, 1043–1052.
[24] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. 2008. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*. 595–604.

Short Paper

Preserving Diversity in Anonymized Data

Mostafa Milani
The University of Western Ontario
London, Ontario, Canada
mostafa.milani@uwo.ca

Yu Huang
McMaster University
Hamilton, Ontario, Canada
huang223@mcmaster.ca

Fei Chiang
McMaster University
Hamilton, Ontario, Canada
fchiang@mcmaster.ca

ABSTRACT

Recent privacy legislation has aimed to restrict and control the amount of personal data published by companies and shared with third parties. Much of this real data is not only sensitive requiring anonymization but also contains characteristic details from a variety of individuals. This diversity is desirable in many applications ranging from Web search to drug and product development. Unfortunately, data anonymization techniques have largely ignored diversity in its published result. This inadvertently propagates underlying bias in subsequent data analysis. We study the problem of finding a diverse anonymized data instance where diversity is measured via a set of diversity constraints. We formalize diversity constraints, and present a clustering-based algorithm for finding a diverse anonymized instance. We show the effectiveness and efficiency of our techniques against existing baselines. Our work aligns with recent trends towards responsible data science by coupling diversity with privacy-preserving data publishing.

1 INTRODUCTION

Organizations often share user information with third parties to analyze collective user behaviour and for targeted marketing. Protecting user privacy is critical to safeguard personal and sensitive data. The European Union General Data Protection Regulation (GDPR), and variants such as the California Consumer Protection Act (CCPA) aim to control how organizations manage user data. For example, a major tenet in GDPR is *data minimization* that states companies should collect and share only a minimal amount of personal data sufficient for their purpose. Given the impossibility of knowing how a published data instance will be used in the future, determining a minimal amount of personal data to share is a challenge.

Privacy-preserving data publishing (PPDP) safeguards individual privacy while ensuring the published data remains practically useful for analysis. Anonymization is the most common form of PPDP, where quasi-identifiers and/or sensitive values are obfuscated via suppression or generalization [11]. As anonymized instances are shared with third parties for decision making and analysis, there is growing interest to ensure that data (and the algorithms that generate and use the data) are diverse and fair. Diversity is a rather established notion in data analytics that refers to the property of a selected set of individuals. Diversity requires the selected set to have a minimum representation from each group of individuals [9, 23] while determining the minimum bound for each group is often domain and user dependent.

To avoid biased decision making, incorporating diversity into computational models is essential to prevent and minimize discrimination against minority groups. In this paper, we focus on diversity, and study how diversity requirements can be modeled and satisfied in PPDP. In PPDP, non-diverse data instances that obfuscate characteristic attributes of a minority group give an inaccurate representation of the population in subsequent data analysis. Unfortunately, early PPDP work [11, 22, 24], and recent work on PPDP for graphs [12, 13], and interactive settings [14, 15] have not considered diversity in published instances.

Example 1.1. Table 1 shows relation R containing patients' medical records describing gender (GEN), ethnicity (ETH), age (AGE), province (PRV), city (CTY), and diagnosed disease (DIAG). Third-parties such as pharmaceuticals, insurance firms are interested in an anonymized R containing patients from diverse geographies, gender, and ethnicities. Let GEN, ETH, AGE, PRV, CTY be quasi-identifier (QI) attributes, and let DIAG be a sensitive attribute. Existing PPDP methods such as k-anonymity prevent re-identification of an individual along the QI attributes from $k-1$ other tuples. Table 2 shows a k-anonymized instance for $k=3$ where tuples are clustered along the QI attributes via value suppression [22, 24].

The k-anonymization problem is to generate a k-anonymous relation through an anonymization process, such as generalization and suppression, while incurring minimum information loss. Suppression replaces some QI attribute values with $\star$s to achieve k-anonymity, and is often considered to be a maximal form of generalization that obscures a value completely. There are several measures of information loss [7, 11], e.g., counting the number of $\star$s. Existing k-anonymization techniques do not preserve diversity in R since these measures do not capture diversity. □

Unfortunately, existing methods fail to provide any diversity guarantees in published, privatized data instances, leading to inaccurate and biased decision making. For example, in Table 2, we have lost the African and Caucasian ethnicity from the (second) group of Male, and the Female gender from the (first) group of Caucasian. These records, which exclude characteristic features of minority groups, misrepresent the true patient population. Efforts to obtain a diverse instance of patients, for example, to obtain minimum proportions of patients along GEN and ETH attributes, are hindered due to these missing values.

To model diversity, existing work has proposed declarative methods in the form of *diversity constraints*, which define the expected frequencies that characteristic values (of a group) must satisfy [23]. Similar to previous work, we consider one or more discrete value attributes to be of particular concern, and we define diversity with respect to the values of these attributes. Using k-anonymity as our privacy definition, and given a relation R, constant k, and a set of diversity constraints Σ, we study the problem of publishing a k-anonymized and diverse instance R^*. An example of a diversity constraint $\sigma_1 = (ETH[Asian], 2, 5)$ requires an anonymized instance to contain between two and five Asian individuals, which is satisfied by Table 1 and Table 2. Diversity constraints provide a declarative definition of the minimum and maximum frequency bounds that specific attribute domain values should appear in R^* [23].

ID	GEN	ETH	AGE	PRV	CTY	DIAG
t_1	Female	Caucasian	80	AB	Calgary	Hypertension
t_2	Female	Caucasian	32	AB	Calgary	Tuberculosis
t_3	Male	Caucasian	59	AB	Calgary	Osteoarthritis
t_4	Male	Caucasian	46	MB	Winnipeg	Migraine
t_5	Male	African	32	MB	Winnipeg	Hypertension
t_6	Male	African	43	BC	Vancouver	Seizure
t_7	Male	Caucasian	35	BC	Vancouver	Hypertension
t_8	Female	Asian	58	BC	Vancouver	Seizure
t_9	Female	Asian	63	MB	Winnipeg	Influenza
t_{10}	Female	Asian	71	BC	Vancouver	Migraine

Table 1: Medical records relation (R)

ID	GEN	ETH	AGE	PRV	CTY	DIAG
r_1	⋆	Caucasian	⋆	AB	Calgary	Hypertension
r_2	⋆	Caucasian	⋆	AB	Calgary	Tuberculosis
r_3	⋆	Caucasian	⋆	AB	Calgary	Osteoarthritis
r_4	Male	⋆	⋆	⋆	⋆	Migraine
r_5	Male	⋆	⋆	⋆	⋆	Hypertension
r_6	Male	⋆	⋆	⋆	⋆	Seizure
r_7	Male	⋆	⋆	⋆	⋆	Hypertension
r_8	Female	Asian	⋆	⋆	⋆	Seizure
r_9	Female	Asian	⋆	⋆	⋆	Influenza
r_{10}	Female	Asian	⋆	⋆	⋆	Migraine

Table 2: Anonymized relation with $k = 3$

ID	GEN	ETH	AGE	PRV	CTY	DIAG
g_1	Female	Caucasian	⋆	AB	Calgary	Hypertension
g_2	Female	Caucasian	⋆	AB	Calgary	Tuberculosis
g_3	Male	Caucasian	⋆	⋆	⋆	Osteoarthritis
g_4	Male	Caucasian	⋆	⋆	⋆	Migraine
g_5	Male	African	⋆	⋆	⋆	Hypertension
g_6	Male	African	⋆	⋆	⋆	Seizure
g_7	⋆	⋆	⋆	BC	Vancouver	Hypertension
g_8	⋆	⋆	⋆	BC	Vancouver	Seizure
g_9	Female	Asian	⋆	⋆	⋆	Influenza
g_{10}	Female	Asian	⋆	⋆	⋆	Migraine

Table 3: Anonymized relation with $k = 2$.

We define the (k, Σ)-anonymization problem, which seeks an optimal k-anonymous instance R^* that satisfies a set of diversity constraints Σ. We propose the *DIVA* algorithm to compute a DIVerse and Anonymized R^*. *DIVA* integrates anonymization with diversity by applying value suppression to find a k-anonymous instance satisfying a set of diversity constraints.

Contributions. We make the following contributions:

(1) We formalize diversity constraitns in PPDP, and we define the (k, Σ)-anonymization problem that seeks a k-anonymous relation with value generalization that satisfies Σ.
(2) We introduce *DIVA*, a clustering-based algorithm that solves the (k, Σ)-anonymization problem with minimal suppression.
(3) We evaluate the effectiveness and efficiency of our selection strategies over the basic version of *DIVA*. We show that *DIVA* achieves improved performance over existing baselines.

2 PRELIMINARIES

Basic Notations. A *relation* R with a *schema* $\mathcal{R} = \{A_1, ..., A_n\}$ is a finite set of n-ary tuples $\{t_1, ..., t_N\}$. A, B, C refer to single attributes and X, Y, Z as sets of attributes.

Privacy-Preserving Data Publishing. k-anonymity prevents re-identification of an individual in an anonymized data set [22, 24]. Attributes in a relation are either *identifiers* such as SSN that uniquely identify an individual, *quasi-identifier* (QI) attributes such as ethnicity, address, age that together can identify an individual, or *sensitive* attributes that contain personal information.

Definition 2.1 (QI-group and k-anonymity). A relation R is k-anonymous if every tuple in R is in a QI-group with at least k tuples. A QI-group is a set of tuples with the same values in the QI attributes. □

For example, $\{r_1, r_2, r_3\}$, $\{r_2, r_3\}$, $\{r_4, r_5\}$, and $\{r_{10}\}$ are QI-groups in Table 2, and the table is 3-anonymous since every tuple in the table is in one of the QI-groups $\{r_1, r_2, r_3\}$, $\{r_4, r_5, r_6, r_7\}$, and $\{r_8, r_9, r_{10}\}$ with at least 3 tuples. Extensions of k-anonymity include l-diversity, t-closeness, and (X, Y)-anonymity, which provide improved privacy confidence (cf. [11] for a survey). We apply k-anonymity for its ease of presentation, however, our definitions and techniques are extensible to include recent PPDP models.

Suppression. Suppression generates an anonymized relation R' from a relation R by replacing some QI values in R with ⋆. We denote this by $R \sqsubseteq R'$. Suppression clearly causes information loss which is typically measured by the number of ⋆s in R'.

Definition 2.2 (k-anonymization problem [24]). Given R, the k-anonymization problem is to find R^* such that (1) $R \sqsubseteq R^*$; (2) R^* is k-anonymous; and (3) R^* has minimum information loss. □

Diversity Constraints. Diversity constraints are originally proposed for the set selection problem defined as follows [23]. Given a set of N items, each associated with a characteristic attribute and a utility score, the *set selection problem* is to select M items to maximize a utility score subject to diversity constraints. The utility score is the sum of scores of each selected item. Let there be d distinct values of the characteristic attribute and m_i with $i \in [1, d]$ be the number of selected items with each distinct value such that $m_i \in [0, M]$ and $\sum_i(m_i) = M$. A diversity constraint ϕ of the form $floor_i \leq m_i \leq ceiling_i$ specifies upper and lower bounds on m_i, i.e. the number of items with the i-th characteristic value. These constraints ensure representation from each category known as coverage-based diversity. To avoid tokenism, where there is only a single representative from each category, we can increase the lower bound, e.g., $m_i > 1$.

Problem Definition. We apply the concept of diversity constraints as proposed by Stoyanovich et. al [23], and introduce a formal definition of diversity constraints in relational data.

Definition 2.3 (Diversity Constraints). A diversity constraint over a relation schema $\mathcal{R}$ is of the form $\sigma = (A[a], \lambda_l, \lambda_r)$ in which $A \in \mathcal{R}$, $a \in dom(A)$ and λ_l, λ_r are non-negative integers. The diversity constraint σ is satisfied by a relation R of schema $\mathcal{R}$ denoted $R \models \sigma$ if and only if there are at least λ_l and at most λ_r occurrences of the value a in attribute A of relation R. We call $[\lambda_l, \lambda_r]$ the frequency range and $A[a]$ the characteristic (or target) value of σ. A set of diversity constraints Σ is satisfied by R, denoted by $R \models \Sigma$, iff R satisfies every $\sigma \in \Sigma$. □

Diversity constraints can be extended to multiple attributes by replacing $A[a]$ with $X[t]$, where X is a set of attributes and t is a tuple with values from these attributes. This extended diversity constraint $\sigma = (X[t], \lambda_l, \lambda_r)$ is satisfied by R if there are at least λ_l and at most λ_r tuples in R with the same attribute values in t. To validate that R satisfies σ, we can run a query that counts the number of occurrences of the target values t in attributes X of R and then check if this number lies in the frequency range $[\lambda_l, \lambda_r]$. Given a set of diversity constraints Σ, we define our problem.

Definition 2.4 (Problem Statement ((k, Σ)-anonymization)). Consider a relation R, a constant k, a set of diversity constraints Σ. The *(k, Σ)-anonymization problem* is to find a relation R^* where: (1) $R \sqsubseteq R^*$, (2) R^* is k-anonymous, (3) $R^* \models \Sigma$, and (4) R^* has minimal information loss, i.e., a minimum number of ⋆'s. □

3 THE DIVA ALGORITHM

We present the DIVersity and Anonymization algorithm (*DIVA*) that solves the (k, Σ)-anonymization problem. *DIVA* takes as input a relation R, a set of diversity constraints Σ, constant k, and returns a k-anonymous and diverse relation R' that satisfies Σ. *DIVA* work in two phases: (i) *clustering*, by partitioning R into disjoint clusters of size greater than or equal to k, while considering Σ; and (ii) *suppression*, by suppressing a minimal number of QI values in each cluster such that they have the same QI values, and form a QI-group of size $\geq k$. The result is a k-anonymous relation, as every QI-group is of size $\geq k$.

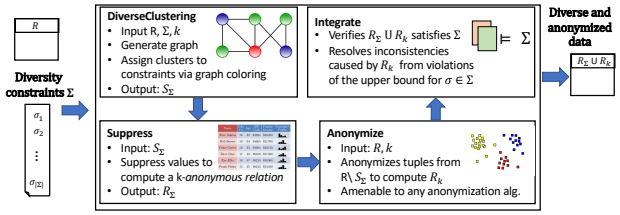


Figure 1: *DIVA* overview.

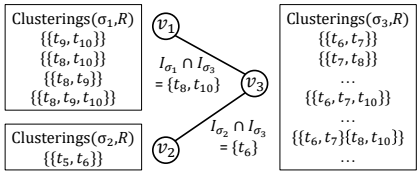


Figure 2: Graph representation of constraints.

3.1 Overview

Figure 1 presents an overview of *DIVA*. The algorithm begins in *DiverseClustering*, which generates a diverse clustering $\mathcal{S}_\Sigma$ that clusters tuples in R such that each $\sigma \in \Sigma$ is satisfied. We note that the clustering $\mathcal{S}_\Sigma$ may involve a subset of tuples in R, which are necessary to satisfy Σ. In *Suppress*, *DIVA* anonymizes the tuples in $\mathcal{S}_\Sigma$ using value suppression. The result is a relation R_Σ that satisfies Σ and is k-anonymous, however, may not represent all tuples in R. In the *Anonymize* step, *DIVA* runs an existing off-the-shelf anonymization algorithm to generate R_k by anonymizing the tuples of R that do not exist in $\mathcal{S}_\Sigma$ and R_Σ. Lastly, the *Integrate* phase integrates R_Σ and R_k to validate that $R_\Sigma \cup R_k$ doesn't violate the constraints' upper bounds.

Algorithm 1 presents *DIVA* details. *DiverseClustering* is a search algorithm that generates the diverse clustering $\mathcal{S}_\Sigma$. For each diversity constraint $\sigma \in \Sigma$, it computes a clustering that ensures the satisfaction of σ. If no diverse clustering exists, i.e., there is no diverse k-anonymous relation R', *DiverseClustering* returns $\mathcal{S}_\Sigma := \emptyset$, and *DIVA* returns an error. Algorithm 2 provides *Suppress* details that takes a clustering $\mathcal{S}$ and returns a relation R_s. For every tuple t in a cluster $C \in \mathcal{S}$, there is a corresponding tuple $r \in R_s$ with the same sensitive values as t. For every QI attribute A_i, $r[A_i]$ is suppressed, i.e. $r[A_i] = \star$, if the tuples in C have different values of A_i (Line 4). Due to this value suppression, R_s contains QI groups corresponding to the clusters in $\mathcal{S}$. In Line 3, we call *Suppress* with input $\mathcal{S}_\Sigma$ and output R_Σ.

Returning to Algorithm 1, there may be tuples of R that do not exist in $\mathcal{S}_\Sigma$. The *Anonymize* routine anonymizes these remaining tuples by applying an existing k-anonymization algorithm to compute R_k. In our evaluation, we use the k-member algorithm [6], but *DIVA* is amenable to any k-anonymization algorithm. Lastly, *Integrate* computes $R' = R_\Sigma \cup R_k$ and checks whether $R' \models \Sigma$. If there exists a violation, R' falsifies the upper bound of some constraint(s) in Σ due to R_k. *Integrate* resolves this by suppressing minimal values in R' to satisfy Σ. We return R' as the final output.

Example 3.1. Consider an execution of *DIVA* with relation R in Table 1, $k = 2$, and $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$, where $\sigma_1 = (ETH[Asian], 2, 5)$, $\sigma_2 = (ETH[African], 1, 3)$, $\sigma_3 = (CTY[Vancouver], 2, 4)$. *DiverseClustering* returns $\mathcal{S}_\Sigma = \{C_1, C_2, C_3\}$ where $C_1 = \{t_9, t_{10}\}$, $C_2 = \{t_5, t_6\}$, and $C_3 = \{t_7, t_8\}$. R_Σ contains suppressed tuples $g_5, ..., g_{10}$ in Table 3 with QI groups $\{g_5, g_6\}$, $\{g_7, g_8\}$, and $\{g_9, g_{10}\}$ that correspond to the clusters in $\mathcal{S}_\Sigma$. R_Σ satisfies Σ where each QI group satisfies a constraint. The *Anonymize* procedure generates

Algorithm 1: *DIVA* (R, Σ, k)

Output: k-anonymous and diverse relation.
1 $\mathcal{S}_\Sigma := DiverseClustering(R, \Sigma, k)$;
2 **if** $\mathcal{S}_\Sigma = \emptyset$ **then return "relation does not exist"**;
3 $R_\Sigma := Suppress(\mathcal{S}_\Sigma)$;
4 **foreach** $C_i \in \mathcal{S}_\Sigma$ **do** $R := R \setminus C_i$;
5 $R_k := Anonymize(R, k)$;
6 **return** $Integrate(R_\Sigma, R_k)$;

Algorithm 2: *Suppress*$(\mathcal{S})$

Input: A clustering $\mathcal{S}$ of tuples with schema $\mathcal{R}$.
Output: Relation R_s (initialized to $\emptyset$).
1 **foreach** $C \in \mathcal{S}$ **and** $t \in C$ **do**
2 $\quad r := t$; $R_s := R_s \cup \{r\}$;
3 $\quad$ **foreach QI Attribute** $A_i \in \mathcal{R}$ **do**
4 $\quad\quad$ **if** $|C[A_i]| > 1$ **then** $r[A_i] := \star$;
5 **return** R_s;

$R_k = \{g_1, ..., g_4\}$, and the *Integrate* procedure returns $R' = R_k \cup R_\Sigma$ (Table 3) which is $k = 2$-anonymous and satisfies Σ. □

3.2 Diverse Clustering

We describe how *DiverseClustering* computes a clustering $\mathcal{S}_\Sigma$ that satisfies Σ. We first define the semantics of how a clustering satisfies a diversity constraint.

Definition 3.2. Given σ defined over R, and a clustering $\mathcal{S}$, $\mathcal{S}$ satisfies σ, denoted as $\mathcal{S} \Vdash \sigma$, if $Suppress(\mathcal{S}) \models \sigma$. The clustering $\mathcal{S}$ satisfies a set of constraints Σ, if $\mathcal{S} \Vdash \sigma_i$ for every $\sigma_i \in \Sigma$. □

Intuitively, $\mathcal{S} \Vdash \sigma$ if the relation returned from $Suppress(\mathcal{S})$ in Algorithm 2 satisfies σ according to Definition 2.3. In Example 3.1, $\mathcal{S}_\Sigma = \{\{t_5, t_6\}, \{t_7, t_8\}, \{t_9, t_{10}\}\} \Vdash \Sigma$ because $Suppress(\mathcal{S}_\Sigma) = \{g_5, ..., g_{10}\} \models \Sigma$.

DiverseClustering finds $\mathcal{S}_\Sigma \Vdash \Sigma$ by computing clusterings $\mathcal{S}_{\sigma_i}$ that satisfy each diversity constraint $\sigma_i \in \Sigma$ and merging them to generate $\mathcal{S}_\Sigma$. In Example 3.1, $\mathcal{S}_{\sigma_1} = \{C_1\} = \{t_9, t_{10}\} \Vdash \sigma_1$, $\mathcal{S}_{\sigma_2} = \{C_2\} = \{t_5, t_6\} \Vdash \sigma_2$, $\mathcal{S}_{\sigma_3} = \{C_3\} = \{t_7, t_8\} \Vdash \sigma_3$, and $\mathcal{S}_\Sigma = \mathcal{S}_{\sigma_1} \cup \mathcal{S}_{\sigma_2} \cup \mathcal{S}_{\sigma_3} = \{C_1, C_2, C_3\} \Vdash \Sigma$. Two conditions must hold while we search for $\mathcal{S}_{\sigma_i}$. First, the clusters in $\mathcal{S}_{\sigma_i}$ must be disjoint, unless they are equal. Specfically, for every pair of clusters $C \in \mathcal{S}_{\sigma_i}$ and $C' \in \mathcal{S}_{\sigma_j}$, either $C' \cap C = \emptyset$ or $C' = C$. For overlapping cluster pairs, the result of *Suppress* will not form QI groups. If we merge the clusters, then the result may not satisfy the conditions. For example, if $\mathcal{S}_{\sigma_2} = \{\{t_5, t_6\}\}$, $\mathcal{S}_{\sigma_3} = \{\{t_6, t_7\}\}$, and $\Sigma = \{\sigma_2, \sigma_3\}$, and we merge them into $\mathcal{S}_\Sigma = \{\{t_5, t_6, t_7\}\}$, then $\mathcal{S}_\Sigma \nVdash \Sigma$, although $\mathcal{S}_{\sigma_2} \Vdash \sigma_2$ and $\mathcal{S}_{\sigma_3} \Vdash \sigma_3$. Second, we select each $\mathcal{S}_{\sigma_i}$ such that it does not falsify the upper bounds of other constraints. In Example 3.1, consider $\Sigma = \{\sigma_2, \sigma_4\}$ with a new constraint $\sigma_4 = (GEN[\text{Male}], 1, 3)$, which requires at least one but not more than 3 men. Then, $\{\{t_5, t_6\}\} \Vdash \sigma_2$, and $\{\{t_3, t_4\}\} \Vdash \sigma_4$, but $\{\{t_5, t_6\}, \{t_3, t_4\}\} \nVdash \{\sigma_2, \sigma_4\}$ since the upper bound of σ_4 is falsified. The clustering $\mathcal{S}_{\sigma_2}$ preserves two more Male values, and falsifying the upper bound in σ_4. This means we cannot build the $\mathcal{S}_{\sigma_i}$ separately, and we must consider the interactions between the constraints. This is clearly a local condition where choosing each $\mathcal{S}_{\sigma_i}$, we consider the related constraints that have overlapping tuples with σ_i, and use this property to convert diverse clustering to graph coloring.

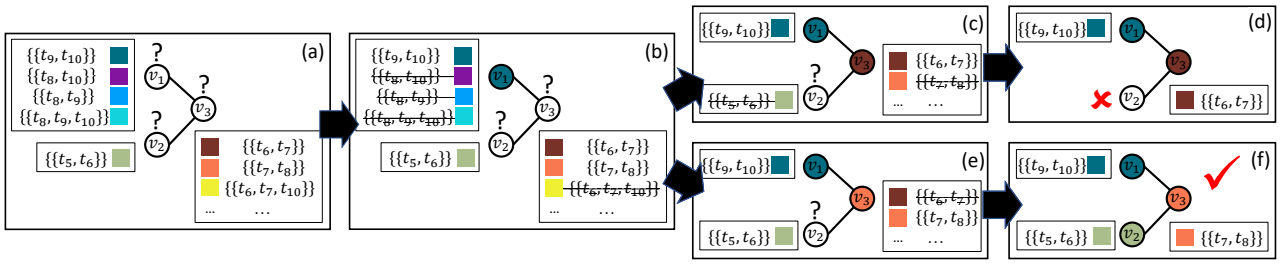


Figure 3: Graph coloring for diverse clustering. Nodes are colored in the order v_1, v_3, v_2.

Algorithm 3: *DiverseClustering*(R, Σ, k)

Output: Clustering $\mathcal{S}_\Sigma$.

1 $G :=$ *BuildGraph*(R, Σ); $V := \emptyset$; $\mathcal{S}_\Sigma := \emptyset$;
2 **if** *Coloring*(G, V, R) **then**
3 | **foreach** $\langle v_i, c_i \rangle \in V$ **do** $\mathcal{S}_\Sigma := \mathcal{S}_\Sigma \cup c_i.clustering$;
4 **return** $\mathcal{S}_\Sigma$;

Algorithm 4: *Coloring*(G, V, R)

Output: true if there is a coloring of G, otherwise **false**.

1 **if** V **contains all nodes of** G **then return true**;
2 $v :=$ *NextNode*(G, V);
3 **foreach** $\mathcal{S} \in$ *Clusterings*$(v.constraint, R)$ **do**
4 | **if** *IsConsistent*$(\mathcal{S}, v)$ **then**
5 | | $c :=$ **new color with clustering** $\mathcal{S}$;
6 | | $V := V \cup \{\langle v, c \rangle\}$;
7 | | **if** *Coloring*(G, V, R) **then return true** ;
8 | | $V := V \setminus \{\langle v, c \rangle\}$;
9 **return false**

3.3 Modeling as Graph Coloring

Given an undirected graph $G = (\Gamma, E)$, where Γ and E denote the set of nodes and edges, respectively, and m distinct colors, the graph coloring problem is to color all nodes subject to certain constraints, e.g., no two adjacent nodes can have the same color. For relation R and diversity constraints Σ, we model each diversity constraint $\sigma_i \in \Sigma$ as a node $v_i \in \Gamma$. We use $v_i.constraint$ to refer to σ_i. An undirected edge $e_{ij} = \{v_i, v_j\} \in E$ exists between nodes v_i and v_j if σ_i and σ_j have overlapping target tuples, i.e., $I_{\sigma_i} \cap I_{\sigma_j}$, where the target tuples of a constraint σ_i, denoted by I_{σ_i}, is the set of tuples in R that have the target values in σ_i.

Example 3.3. Figure 2 shows G with three nodes corresponding to $\sigma_1, \sigma_2, \sigma_3$, and each of their neighboring constraints modeled via edges $E = \{\{v_1, v_3\}, \{v_2, v_3\}\}$. The edge labels show the non-empty intersections between their target tuple sets. The sets of target tuples are $I_{\sigma_1} = \{t_8, t_9, t_{10}\}$, $I_{\sigma_2} = \{t_5, t_6\}$, and $I_{\sigma_3} = \{t_6, t_7, t_8, t_{10}\}$, which means σ_1, σ_3 and σ_2, σ_3 are neighboring constraints, but there is no edge between v_1 and v_2 because $I_{\sigma_1} \cap I_{\sigma_3} = \{t_8, t_{10}\}$ and $I_{\sigma_2} \cap I_{\sigma_3} = \{t_6\}$, and $I_{\sigma_1} \cap I_{\sigma_2} = \emptyset$. Beside each node, we show the clusterings that satisfy the corresponding constraint.

Choosing a color for node v_i is analogous to finding a clustering $\mathcal{S}_{\sigma_i}$ for σ_i. The goal is to color all nodes, while the color of each node is *consistent* with the color of its neighboring nodes. This means the corresponding clusterings must satisfy the two conditions that we mentioned earlier. For a color c, $c.clustering$ refers to its corresponding clustering.

Algorithm 3 presents the details of *DiverseClustering*. We build the graph G for Σ and R (Line 1). We then initialize the clustering $\mathcal{S}_\Sigma$ and a mapping V that stores the color (assigned clustering) for each node, and check if a coloring exists via *Coloring*.

Algorithm 4 presents the recursive function, *Coloring*, that takes a graph G, the mapping V (specifying the colored nodes), relation R, and returns *true* if the remaining nodes of G can be colored; otherwise it returns *false*. Note that choosing a color for a node, can restrict the choice of colors for neighboring nodes when the clusterings have overlap. *Coloring* iterates over every uncolored node and assigns a color (clustering) that is consistent with its neighboring nodes (constraints). Specifically, we check that the two search conditions mentioned earlier are satisfied. We propose three versions of *DIVA*: (1) DIVA-Basic: *Coloring* randomly selects an uncolored node (Line 2) to color using *NextNode*; (2) MaxFanOut: selects constraints with a minimal number of clusterings; and (3) MinChoice: selects constraints with a maximal overlap with neighboring constraints. We describe the latter two versions later in this section.

Given a node v, we color v by checking whether its candidate clustering, and its adjacent nodes are consistent (Alg. 4, Lines 3-8). The *Clusterings* routine returns minimal clusterings $\mathcal{S}$ that satisfy $v.constraint$ ($Suppress(\mathcal{S}) \vDash v.constraint$). For example, $Clusterings(\sigma_1, R)$ contains four clusterings $\{\{t_8, t_9\}\}$, $\{\{t_8, t_{10}\}\}$, $\{\{t_9, t_{10}\}\}$, $\{\{t_8,\ t_9, t_{10}\}\}$, while $Clusterings(\sigma_2, R)$ contains one clustering $\{\{t_5, t_6\}\}$. In Lines 4-8, we check whether $\mathcal{S}$ is consistent with the clusterings of the neighboring constraints. If so, we assign a new color c to the clustering $\mathcal{S}$, and we temporarily color v with c by adding $\langle v, c \rangle$ to V. We then recursively call *Coloring* to check whether the remaining nodes in G can be colored. If color c does not work, i.e. *Coloring* returns false, we remove $\langle v, c \rangle$ from V, and try another color. If all clusterings are inconsistent, i.e., there is no successful coloring of v, we return false (Line 9), to backtrack and evaluate a different node. To compute a satisfying clustering depends not only on k, and the frequency of characteristic values in R with respect to $[\lambda_l, \lambda_r]$ in σ, but also on the distribution of these characteristic values. We empirically study the impact of data distribution on accuracy in Section 4.

Example 3.4. Figure 3 shows an execution of *Coloring* over graph G with nodes $\{v_1, v_3, v_2\}$. Figure 3(a) initializes nodes as uncolored. We first consider v_1, and select $S_{\sigma_1} = \{\{t_9, t_{10}\}\}$ (Figure 3(b)). We color nodes v_2 and v_3 by recursively calling *Coloring*. Coloring one node may restrict the color choice of neighboring nodes, e.g. after we select $\{\{t_9, t_{10}\}\}$ for v_1, we cannot select $\{\{t_6, t_7, t_{10}\}\}$ for v_3 due to the overlapping tuple t_{10}. For node v_3, we have several choices including $\{\{t_6, t_7\}\}$ and $\{\{t_7, t_8\}\}$. In Figure 3(c), we assume the coloring algorithm chooses $\{\{t_6, t_7\}\}$ for v_3. As a result, $\{\{t_5, t_6\}\}$, which was the only choice for v_2, cannot be used due to the overlapping tuple t_6. This leads the algorithm towards an unsatisfying clustering (Figure 3(d)). *DIVA* backtracks its last decision for v_3 by selecting a different color, $\{\{t_7, t_8\}\}$ for v_3 in Figure 3(e). In this case, the clustering $\{\{t_5, t_6\}\}$ for v_2 does

Table 4: Data characteristics.

	Pantheon	Census	Credit	Pop-Syn
$\|R\|$	11,341	299,285	1000	100,000
n	17	40	20	7
$\|\Pi_{QI}(R)\|$	5,636	12,405	60	24,630
$\|\Sigma\|$	24	21	18	10

not overlap with $\{\{t_7, t_8\}\}$. Since we have found a clustering that satisfies all the constraints (i.e., a coloring of all nodes), *Coloring* returns *true* with V containing the nodes and their colors (i.e., clusterings). *DiverseClustering* uses V to compute the final clustering as $\mathcal{S}_\Sigma = \{\{t_5, t_6\}, \{t_7, t_8\}, \{t_9, t_{10}\}\}$. □

DIVA runs in polynomial time w.r.t. the size of R. The runtime is split between *DiverseClustering* and *Anonymize*. *DiverseClustering* runs in polynomial time as the number of clusters considered in coloring for each constraint is polynomial w.r.t. R. *Anonymize* runs an off-the-shelf algorithm. We use the k-member anonymization algorithm, which runs in polynomial time [6].

Selection Strategies. In the basic version of *DIVA*, we randomly select a constraint and a clustering to evaluate. These choices impact performance as poor initial selections can lead to increased backtracking operations downstream. We selectively order the constraints (nodes) and clusterings (colors) that most likely lead to a graph coloring while minimizing the need to backtrack.

MinChoice: We select the most restrictive constraints first, i.e., in *Nextnode*, we select v with minimum $|Clusterings(v.constraint, R)|$. As we visit nodes and assign (colors) clusterings, we update the candidate clusterings for their neighbors.

MaxFanOut: We select constraints with maximum overlap with other constraints, i.e, nodes with the maximum number of uncolored neighbors. We preferentially select these constraints due to their high number of interactions, which lead to an increased number of target attributes, and bounds that the relevant tuples must satisfy. This heuristic aims to prune unsatisfiable clusterings early to reduce the number of clustering evaluations downstream.

4 EXPERIMENTS

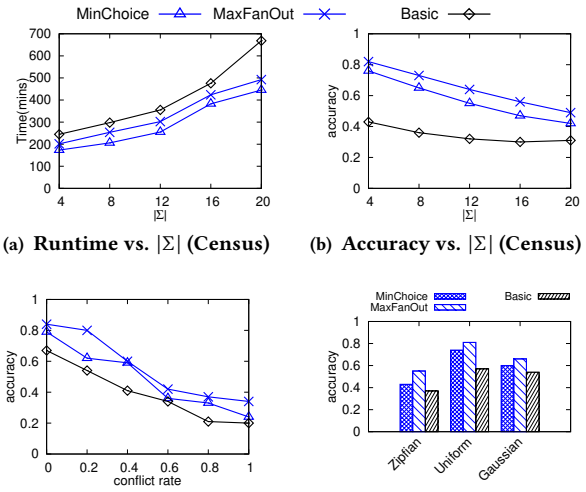
We evaluate the accuracy and performance of *DIVA*, and compare against three k-anonymization baselines to evaluate the cost of incorporating diversity constraints.

Experimental Setup. We implement *DIVA* using Python 3.6 on a server with 32 Core Intel Xeon 2.2 GHz processor with 32GB RAM. We use three real datasets: (1) *Pantheon* describes individuals on Wikipedia [1]; (2) *Census* from the U.S. Census Bureau describes population data [3]; and (3) *German Credit* describes credit risk according to demographic attributes [3]. We also generate a synthetic population dataset, *Pop-Syn* to specify population characteristics using Synner.io [18]. Table 4 summarizes the dataset characteristics. We implement three notions of diversity via three classes of diversity constraints, namely, minimum frequency, average, and proportional representation from the attribute domain [23]. We found proportion constraints capture the relative distribution in the attribute domain with less sensitivity than average, and run our experiments using proportion constraints. The set of defined constraints, datasets and our source code are available at [2, 19].

Metrics and Parameters. We compute the average runtime over five executions. To quantify accuracy, we seek anonymizations with indistinguishable tuples, and minimize information loss. We use the *discernibility metric*, $disc(R', k)$, which assigns a penalty to each tuple based on the number of tuples that are indistinguishable from it in R' for a given k, reflecting its information loss [4]. We measure the *conflict rate* between a pair of diversity constraints as the number of overlapping relevant tuples, and we extend this definition to a set of diversity constraints. Conflict values range from [0,1], where 0 indicates no overlap. The discernibility metric and conflict rate definitions can be found in [19]. Table 5 lists parameter ranges and default values.

Table 5: Parameter values (defaults in bold)

Symbol	Description	Values
$\|R\|$	#tuples	60k, 120k, **180k**, 240k, 300k
$\|\Sigma\|$	#constraints	4, **8**, 12, 16, 20
$cf(\Sigma)$	conflict rate	0, **0.2**, 0.4, 0.6, 0.8, 1
k	minimum cluster size	10, **20**, 30, 40, 50



(a) **Runtime vs.** $|\Sigma|$ **(Census)** (b) **Accuracy vs.** $|\Sigma|$ **(Census)**

(c) **Accuracy vs.** cf **(Pantheon)** (d) **Acc. vs. distrib. (Pop-Syn)**

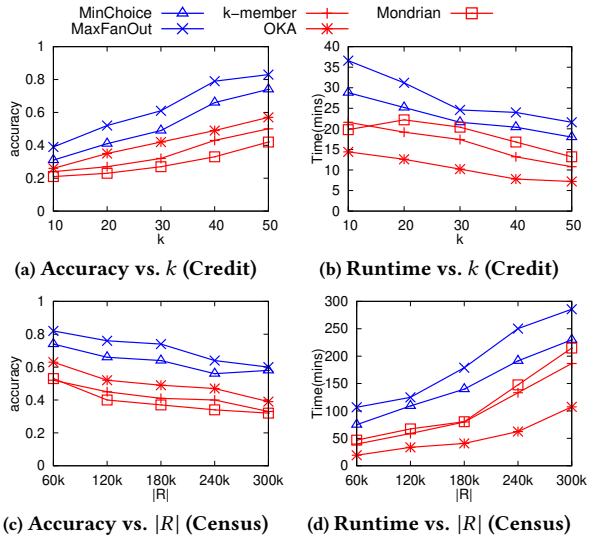
Figure 4: *DIVA* **efficiency and effectiveness.**

4.1 Accuracy and Performance

Figure 4a and 4b show *DIVA* runtime and accuracy, respectively, as we vary $|\Sigma|$. DIVA-Basic shows exponential growth in runtime since we can assign $O(|R|)$ different clusterings to each constraint. Our selection strategies to restrict clusterings and perform early pruning, MinChoice and MaxFanOut, show linear scale-up. Figure 4b shows as $|\Sigma|$ increases, we see accuracy decline at a linear rate. As new $\sigma \notin \Sigma$ are added, we observe new relevant tuples join existing clusters of relevant tuples from Σ leading to a smaller decline in accuracy. This occurs with multi-attribute constraints that share target attributes with single attribute constraints. The alignment of QI and target attribute values between new and existing tuples influence the rate of decline.

Figure 4c shows *DIVA* accuracy as we vary conflict rate, cf. As expected, accuracy declines for increasing cf, with MaxFanOut and MinChoice outperforming DIVA-Basic by +17% and +9%, respectively. MaxFanOut outperforms MinChoice since targeting constraints with a high number of interactions eliminates unsatisfying clusterings sooner, while satisfying dependent constraints.

To measure accuracy for different data distributions, we generate attribute values according to the Zipfian, uniform, and Gaussian distributions with $|R| = 100k$, $|\Sigma| = 8$. Figure 4d shows that MaxFanOut performs best across all distributions by 8% and 17% over MinChoice and DIVA-Basic, respectively. The target uniform distribution performs best as domain values are spread evenly across the tuples, avoiding contention among a small set



(a) **Accuracy vs. k (Credit)** (b) **Runtime vs. k (Credit)**

(c) **Accuracy vs. $|R|$ (Census)** (d) **Runtime vs. $|R|$ (Census)**

Figure 5: Comparative study: anonymization baselines.

of tuples. This conflict occurs more often in the Zipfian case than the Gaussian, leading to lower accuracy.

4.2 Baseline Comparison

Figure 5a and Figure 5b show the comparative accuracy and runtime, respectively, of *DIVA* against three baseline k-anonymization algorithms: k-member [6], OKA [17], and Mondrian [16]. MinChoice and MaxFanOut incur higher runtimes reflecting the cost of computing a *diverse* data instance. We expect this to be acceptable in practice since constraint validation and anonymization is often done offline. As k increases, we expect more tuples to be anonymized leading to higher penalty costs. We note that runtimes decrease for increasing k as more values are suppressed to satisfy k-anonymity on each cluster. This improves the efficiency of consistency checking in the coloring algorithm since we can prune clusterings with size less than k during backtracking.

Figures 5c and 5d show that *DIVA* and the baselines are sensitive to $|R|$. Figure 5c shows that *DIVA* achieves improved accuracy over the baseline algorithms, in addition to satisfying diversity constraints. As new attribute values are introduced, they may not align with existing values in the clusters, requiring further suppression, and decreasing accuracy. In Figure 5d, all techniques incur increased runtimes due to evaluating a larger number of clusters. For *DIVA*, a larger number of tuples and clusters also increases the likelihood of potential conflicts among clusterings.

5 RELATED WORK

Privacy Preserving Data Publishing. Extensions of k-anonymity include *l-diversity*, *t-closeness*, and *(X,Y)-anonymity* with tighter privacy guarantees [11]. *DIVA* is extensible to re-define the clustering criteria according to these privacy semantics. Our empirical study has shown that *DIVA* generates anonymized instances comparable to existing baselines, but also guarantees diversity requirements are satisfied.

Fairness and Diversity. Data sharing of private data has been studied along two primary lines. First, causality reasoning aims to recognize discrimination to achieve algorithmic transparency and fairness. Recent techniques have proposed influence measures to identify correlated attributes [8], and statistical reasoning about discrimination [20]. Secondly, recent work have studied variants of DP to release synthetic data with similar statistical properties to the input data [5], and studying the impact of DP algorithms on equitable resource allocation [21]. Our work is complementary but with a different goal; to publish diverse and anonymized versions of the original data with minimal information loss for applications where statistical summaries, synthetic data, and aggregate queries are inadequate. Recent work by Stoyanovich et. al., study diversity in the set selection problem and introduce diversity constraints to guarantee representation in the selected set [23, 25]. We build upon this work, and are the first to formalize diversity constraints. Our algorithms couple diversity with data anonymization, a problem not considered in existing work.

6 CONCLUSION AND FUTURE WORK

We formalize diversity constraints, and introduce *DIVA*, a DIVersity-driven Anonymization algorithm that computes a privatized data instance satisfying a set of diversity constraints. Our evaluation show the performance benefits of our optimizations, and the overhead of enforcing diversity constraints over baselines. As future work, we intend to study privacy extensions beyond k-anonymity, e.g. randomization algorithms to satisfy both diversity constraints and *Differential privacy (DP)* to provide a higher level of protection for individuals [10]. We also intend to explore a distributed version of the coloring algorithm to improve scalability by satisfying constraints in parallel.

REFERENCES

[1] 2014. *Pantheon Dataset.* https://pantheon.world/
[2] 2020. *DIVA: Extended Evaluation.* https://diva1234567.github.io/DIVA/
[3] 2020. *UCI ML Repository.* https://archive.ics.uci.edu/ml/datasets/
[4] R. Bayardo and R. Agrawal. 2005. Data Privacy through Optimal K-Anonymization. In *ICDE*. 217–228.
[5] V. Bindschaedler, R. Shokri, and C. Gunter. 2017. Plausible Deniability for Privacy-Preserving Data Synthesis. *PVLDB* 10, 5 (2017), 481–492.
[6] J. Byun, A. Kamra, E. Bertino, and N. Li. 2007. Efficient k-anonymization using clustering techniques. In *DASFAA*. 188–200.
[7] F. Chiang and D. Gairola. 2018. InfoClean: Protecting Sensitive Information in Data Cleaning. *Journal of Data and Information Quality* 9, 4 (2018), 22:1–22:26.
[8] A. Datta, S. Sen, and Y. Zick. 2016. Algorithmic Transparency via Quantitative Input Influence: Theory and Experiments with Learning Systems. In *Symposium on Security and Privacy*. 598–617.
[9] M. Drosou, H. Jagadish, E. Pitoura, and J. Stoyanovich. 2017. Diversity in Big Data: A Review. *Big Data* 5, 2 (2017), 73–84.
[10] C. Dwork. 2006. Differential Privacy. In *International Colloquium on Automata, Languages and Programming*. 1–12.
[11] B. Fung, K. Wang, R. Chen, and P. Yu. 2010. Privacy-Preserving Data Publishing: Survey of Recent Developments. *ACM Comput. Surv.* 42, 4 (2010).
[12] B. Grau and E. Kostylev. 2016. Logical Foundations of Privacy-preserving Publishing of Linked Data. In *AAAI*. 943–949.
[13] M. Hay, G. Miklau, D. Jensen, D. Towsley, and C. Li. 2010. Resisting structural reidentification in anonymized social networks. *VLDB J.* 19, 6 (2010), 797–823.
[14] Y. Huang, M. Milani, and F. Chiang. 2018. PACAS: Privacy-Aware, Data Cleaning-as-a-Service. In *International Conference on Big Data*. 1023–1030.
[15] Y. Huang, M. Milani, and F. Chiang. 2020. Privacy-aware data cleaning-as-a-service. *Information Systems* 94 (2020), 101608.
[16] K. LeFevre, D. DeWitt, and R. Ramakrishnan. 2006. Mondrian multidimensional k-anonymity. In *ICDE*. IEEE, 25–25.
[17] J. Lin and M. Wei. 2008. An efficient clustering method for k-anonymization. In *PAIS*. 46–50.
[18] M. Mannino and A. Abouzied. 2019. Is This Real? Generating Synthetic Data That Looks Real. In *UIST*. 549–561.
[19] M. Milani, Y. Huang, and F. Chiang. 2020. Diversifying Anonymized Data with Diversity Constraints. *arXiv preprint arXiv:2007.09141* (2020).
[20] R. Nabi and I. Shpitser. 2018. Fair Inference on Outcomes. In *AAAI*. 1931–1940.
[21] D. Pujol, R. McKenna, S. Kuppam, M. Hay, A. Machanavajjhala, and G. Miklau. 2020. Fair Decision Making Using Privacy-Protected Data. In *FAccT*. 189–199.
[22] P. Samarati. 2001. Protecting respondents identities in microdata release. *TKDE* 13, 6 (2001), 1010–1027.
[23] J. Stoyanovich, K. Yang, and H. Jagadish. 2018. Online set selection with fairness and diversity constraints. In *EDBT*. 241–252.
[24] L. Sweeney. 2002. k-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.* 10, 05 (2002), 557–570.
[25] K. Yang and J. Stoyanovich. 2017. Measuring Fairness in Ranked Outputs. In *SSDBM*. 22:1–22:6.

Short Paper

Automatic Tuning of Read-Time Tolerances for Optimized On-Demand Data-Streaming from Sensor Nodes

Julius Hülsmann
Technische Universität Berlin
j.huelsmann@tu-berlin.de

Chiao-Yun Li
Fraunhofer FIT
chiao-yun.li@fit.fraunhofer.de

Jonas Traub
Technische Universität Berlin
jonas.traub@tu-berlin.de

Volker Markl
Technische Universität Berlin
volker.markl@tu-berlin.de

ABSTRACT

The Internet of Things provides applications with data streams from billions of sensor devices in real-time. Usually, sensor devices serve multiple queries simultaneously despite having limited computational capabilities. This paper presents a solution for reducing the number of data reads and transmissions by increasing the potential for sharing reads among concurrent streaming queries. Existing read-scheduling techniques on sensor nodes dynamically adjust the data-acquisition rate depending on the data's variability. However, they leave the definition of read-time tolerances to the user. Such read-time tolerances are crucial for sharing reads among queries. We extend previous work by presenting a generally-applicable algorithm that defines read-time tolerances and adapts them on-the-fly depending on observed data characteristics. We evaluate our solution on real-world data and show that it reduces the sensing error by up to 60% compared to existing approaches with the same number of data reads. Respectively, our technique reduces the number of data reads to achieve the same sensing error as existing techniques. To the best of our knowledge, we are the first to automatically set and tune read-time tolerances to reduce sensor readings and data transmissions on sensor nodes.

1 INTRODUCTION

With the advance in communication and information technologies, we can easily access the internet with laptops, tablets, smartphones, and other mobile devices. The network of smart devices, vehicles, and other network-attached sensors forms the Internet of Things (IoT) [16]. In a general IoT setup, multiple sensors are attached to a single device, the sensor node. The sensor node gathers data from the attached sensors and then transmits them to the Stream Processing Engine, enabling it to answer queries.

IoT applications make decisions in real-time based on dynamically changing surroundings. With a vast amount of sensors and the need for real-time data processing, several efficient data acquisition and transmission techniques are proposed to provide data streams to applications [8, 11, 21, 22, 26].

There exist several solutions for making data acquisition and transmission from sensor nodes more efficient. These solutions aim at reducing the number of sensor reads conducted (adaptive sampling) or values transmitted (adaptive filtering) [8]. The key idea of *adaptive sampling* is to modify the read-time frequency based on the recent history of sensor readings. If the values read from the sensor behave unexpectedly (i.e., have a high variability), samples are collected at a higher rate. In contrast, if values barely change, the sampling rate is reduced. *Adaptive filtering* techniques reduce the number of transmitted values by discarding values that evolve predictably. Adaptive sampling and adaptive filtering usually operate on a per-query basis. Instead, *multi-query read-sharing* saves transmissions by scheduling reads that satisfy multiple concurrent queries simultaneously.

Adaptive sampling and adaptive filtering techniques have to be selected and configured in accordance with the corresponding query's *data demand*, i.e., reflecting the consumer's sensitivity to observing changes in the data. It is crucial to note that queries may possess different data demands. For example, the adaptive sampling technique AdaM [23] reacts very fast to abrupt value changes, while FAST [6] incorporates differential privacy features. This is where multi-query optimization becomes necessary.

In previous works, we propose considering the queries' data demand during multi-query optimization by combining adaptive sampling and filtering techniques with multi-query read sharing in a unified framework [9, 20]. User-defined stateful functions (e.g., adaptive sampling techniques) iteratively suggest a query's read-times, and a multi-query read-scheduler exploits read-time tolerances around suggested read-times for sharing reads among queries respecting their data demand.

Figure 1 shows an overview of these two steps of read-time suggestion and read-fusion. As we can see from steps ② and ③, specifying read-time tolerances is essential to read-fusion. However, it is hard for users to manually specify and tune such tolerances. While well-known adaptive sampling techniques provide the desired read-time, they do not define tolerances. Ideally, tolerances for each read request should adjust automatically based on current data characteristics. Intuitively, tolerances should increase if values remain constant or follow an expected trend. Tolerances should shrink if sensor values change rapidly.

In this paper, we introduce an Adaptive Read-Time Tolerance Controller (ARTC), a general algorithm that adjusts read-time tolerances on-the-fly, based on the recent history of sensor readings. ARTC enhances arbitrary adaptive sampling techniques with automated control of read-time tolerances. Such read-time tolerances enable the sharing of sensor values among multiple queries on the same sensor node and thus lead to reading and transmission savings in distributed sensor networks. We design our solution to be generally applicable – independent of the algorithm defining the desired read-times. Therefore, we divide the task of setting read-time tolerances into two parts: Iteratively adapting the read-time tolerance's diameter, and shifting the read-time tolerance interval around the desired read-time: We capture statistics on the data's variability based on which we adapt the read-time tolerance. We use a proportional-integral-derivative controller (PID controller) [2, 3], which is a commonly used form of feedback control [4] to shift the read-time tolerance.

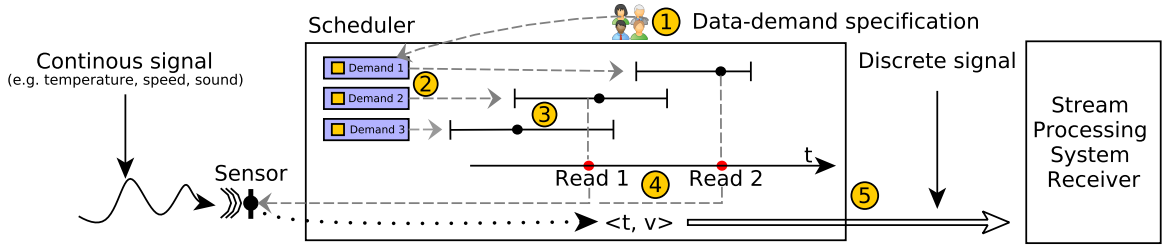


Figure 1: Overview of Optimized On-Demand Data-Streaming from Sensor Nodes [20]: ① The sensor node's scheduler receives the *data demand* of new queries. ② A query's *data demand* defines the next (desired) read-time • at which it requires the next sensor reading and optionally, a read-time tolerance. ③ The scheduler determines the time for the next sensor reading •, fusing requests, if possible. The device wakes up to ④ read a value from the sensor, and ⑤ transmits it to the receivers.

Our implementation is available as an open-source project[1]. We evaluate our solution by simulating sensor sharing based on multiple real-world IoT datasets. We analyze the error and the read-time tolerance induced by read-sharing for different configurations of our algorithm, compare it to setups with fixed read-time tolerances, and show that our solution outperforms fixed read-time tolerance specifications.

The remainder of this paper is structured as follows: We first present our solution in Section 2, which we evaluate in Section 3 on three real-world datasets. After discussing Related Work in Section 4 we conclude in Section 5.

2 ADAPTIVE READ-TIME TOLERANCE

This section explains how our adaptive read-time tolerance controller ARTC adapts the read-time tolerance to the sensed data's variability. We first showcase how to use ARTC in Section 2.1 and subsequently present our solution's internals in Section 2.2.

2.1 The User's Perspective

The user configures ARTC with two parameters: Firstly, E, indicating the permissible inaccuracy in the data-representation relative to the data's average magnitude. Secondly, the optional parameter `dMax`, which specifies a maximal read-time tolerance.

It is essential to highlight that the complexity of using ARTC is transparent to users. They can define streaming queries in declarative languages to the Stream Processing Engine (SPE). The user parametrizes ARTC as part of the query as sketched below:

```
SELECT [t, speed, position]    // SENSOR IDENTIFIERS
FROM [bus 1]                   // SENSOR NODE IDENTIFIER
USING [AdaM]          // READ-TIME SUGGESTION ALGORITHM
WITH [ARTC(E=0.1)] ON [speed]   // CONFIGURATION ARTC
```

As outlined in Figure 1, users can submit queries to the SPE, which then forwards the data demand specification and involved sensors to the specified sensor device ①. The device can thus schedule reads according to the specified data-requirements ③ and to dispatch the data stream to the user ⑤. The user receives a data stream $(t_i, \text{speed}_i, \text{position}_i)_{i \in \mathbb{N}}$ until terminating the query.

2.2 Architecture

We first show how our solution integrates within the multi-query read-scheduling framework executed on the sensor device [20] and then present the internals of ARTC. Table 1 summarizes the newly introduced nomenclature.

[1]https://github.com/TU-Berlin-DIMA/ARTC

Internal Architecture. Once the sensor device receives at least one query from the SPE, it schedules reads, as shown in Algorithm 1. For each new query, the sensor device first performs a read in Line 4. The sensor device then reports the value v gathered from the sensor at time t back to the client who issued the query. It then feeds the read-time and value into the read-time suggestion algorithm (Line 7), which generates the next desired read-time t_D. In the example query in Section 2.1, the read-time suggestion is performed through the adaptive sampling algorithm AdaM. Then, we forward the desired read-time and the last sensor reading to the read-time tolerance algorithm (Line 7), which computes a read-time tolerance by specifying interval boundaries t_s and t_e that enclose the desired read-time. Together with the desired read-time t_D, the interval boundaries constitute the next read request of the query. Lastly, the multi-query read-scheduler schedules reads according to all read requests. We refer the reader to our previous work for details on the optimizations the multi-query read scheduler performs and different optimization objectives [20]. The sensor device reduces its energy consumption by then sleeping until the next read is due. The process of updating the read request is repeated for all queries which received the last sensor reading.

Adaptive Read-Time Tolerance Controller (ARTC). A read-time tolerance algorithm has to meet the following requirements: (i) Despite advances in hardware technologies, sensor devices are still restricted in computational capabilities compared to sinks. As they need to schedule reads very precisely and are often battery-powered, the algorithm has to be energy-efficient. (ii) The algorithm has to adapt to changes in the distribution of the data quickly. Otherwise, the user misses important events. (iii) The algorithm has to be easy to configure and should not be entirely dependent on the read-time suggestion algorithm.

As a lightweight means of keeping track of the distribution of gathered sensor values, we use the iterative Probabilistic Exponentially Weighted Moving Average (PEWMA) algorithm. We use two instances of PEWMA; firstly, we monitor the distribution of the magnitude of sensor readings $\|v\|_2$. We refer to the probabilistic moving average by μ_v and to the estimated standard deviation of the used normal distribution by σ_v. We use the second instance of PEWMA to monitor moving average μ_δ and standard deviation σ_δ of the difference between consecutive read-times $\delta := \|v - v_l\|_2$. In order to be less dependent on the read-time suggestion algorithm, we divide our solution into two parts: (1) the adaptation of the read-time tolerance Δ based on the predictability of the data distribution and (2) shifting of the read-time interval $s \in [-0.5, 0.5]$ based on the value deviation between consecutive sensor readings as shown in Equation 1:

Algorithm 1: Multi-query read-sharing algorithm.

```
t = now();  involvedQueries = newQueries;  u = {};
while running do
    sleepUntil(t);
    v = sensor.read();
    involvedQueries.forwardToSink(t, v);
    while query in involvedQueries do
        t_D = readTimeSuggestion.next(t, v);
        t_s, t_e = readTimeToleranceAlgorithm.next(t, v, t_D);
        u[query] = (t_s, t_D, t_e);
    end
    t, involvedQueries = mqrs.select(u);
end
```

$$t_s = t_D + \Delta \cdot \left(s - \frac{1}{2}\right), \qquad t_e = t_D + \Delta \cdot \left(s + \frac{1}{2}\right) \tag{1}$$

In the next sections, we discuss separately how read-time tolerances Δ and shifts s are computed.

Adaptation of read-time tolerance. We use the variation of δ to assess the predictability of the distribution. If the variation exceeds the threshold E specified by the user,

$$\begin{aligned} & \sigma_\delta && > E \cdot \mu_v \\ \Leftrightarrow \quad & \sigma_\delta - E \cdot \mu_v && > 0, \end{aligned} \tag{2}$$

we reduce the read-time tolerance Δ by the damping factor in Equation 3. This way, it decreases exponentially in the number of consecutive times that the variation exceeds E.

$$\Delta \leftarrow \Delta \cdot \frac{1}{2}. \tag{3}$$

Otherwise, we increase the read-time tolerance by a value proportional to the difference between the user-indicated deviation E and σ_δ, which we denote the step-size δ_E. In order to make the step-size independent of the magnitude of both E and the sensor values, we scale the threshold by the multiplicative inverse of E and μ_v and define the step size in Equation 4.

$$\delta_E := \frac{\sigma_\delta - E \cdot \mu_v}{E \cdot \mu_v} = \frac{\sigma_\delta}{E \cdot \mu_v} - 1. \tag{4}$$

Shifting of read-time interval. We shift the read-time interval around the desired read-time t_D to reduce the algorithm's dependence on the read-time suggestion algorithm's performance. We use a PID controller with `setpoint` E to assess whether the deviation between consecutive sensor readings is within the range specified by the user. That way, we obtain a long-term estimation of the read-time suggestion algorithm's performance and exert limited control over the read-time difference. We scale the controller's output with a factor of 0.1 we determined empirically, which achieves good results for all evaluated datasets.

Complete Algorithm. The overall algorithm is summarized in Algorithm 2: We restate Condition 2 in terms of δ_E and also ensure the user-defined boundaries of Δ and the range of s.

3 EXPERIMENTAL EVALUATION

In this section, we evaluate ARTC on three real-world IoT datasets. We picked the different datasets to portray various data characteristics, which we lay out in Section 3.1. We then introduce our experimental setup in Section 3.2, present the experiments and their results in Section 3.3, and close with a discussion in Section 3.4.

Variable	Description
t,t_l,v,v_l	Current / last sensor reading (time and value).
t_s,t_e	Read-time tolerance interval boundaries.
$t_D \in [t_s,t_e]$	Desired read-time.
δ	eukl. distance between consecutive values.
μ_v,σ_v μ_δ,σ_δ	PEWMA and estimated standard deviation of the magnitude of v, respectively δ.

Table 1: Overview of subsequently used nomenclature.

PEWMA	AdaM	PID Controller	ARTC
$\alpha = 0.5,\ \beta = 0.5,$ $d_{\text{init}} = 20$	$\gamma = 0.2$	$P = 2, I = 2e\text{-}3,$ $D = 0.3$	dMax = 175

Table 2: Configuration of AdaM's and ARTC's components.

3.1 Datasets

We evaluate our solution on three IoT sensor datasets [17]. We provide a brief description for each dataset and state which of the multiple sensors in the dataset we use for our experiments.

The **football monitoring dataset [14]** provided within the scope of the ACM DEBS 2013 Grand Challenge consists of data gathered during a football training game at the Nuremberg Stadium in Germany. We replay the football's absolute velocity in m/s, which is available at 2000 Hz for the match's first half-hour. We evaluate our solution on this dataset as it is very volatile, and the speed of the football spikes abruptly when kicked by a player.

The **daily and sports activities dataset [1]** contains multiple time-series, constituting different individuals performing a total of 19 activities such as sitting, standing, and jumping. We replay data from the torso acceleration sensor in m/s^2, which is available at 25 Hz frequency for 5 minutes per person and activity. We select two daily activities sitting and standing, and two sports activities descending stairs and exercising on a stepper, and concatenate the corresponding time-series. As different activities alternate in the resulting time series, we can observe a drastic shift in the distribution of values each time the performed activity changes.

The **gas dataset [7]** holds concentration levels of dynamic gas mixtures. The data is collected continuously at a frequency of 100 Hz, and we use the first hour of the available data. We replay the gas mixture of ethylene and CO, and the unit of the measurement is *parts per million* (ppm). The variability in this dataset is low, as the distribution of the different chemicals only changes gradually.

3.2 Evaluation Setup

In a production setup, a read-sharing algorithm's performance on a sensor device is measured by counting the number of saved reads of the corresponding query under heavy load. However, this quantity largely depends on the configuration and number of concurrent queries. To assess the performance of read-sharing more objectively, we measure the suggested read-time tolerance Δ instead, while operating only a single query. We simulate heavy load by sampling at random within the proposed read-time tolerance to obtain a realistic view of the induced error. For each experiment, we conduct multiple runs and combine the results.

We record two performance measures during the conducted experiments to evaluate the functionality and effectiveness of ARTC. Firstly, we keep track of the average read-time tolerance μ_Δ, which indicates the read-sharing potential enabled by the algorithm for a total of M suggested read-time tolerances Δ_i: Secondly, we record the induced error for each point in time t, i.e., the distance between the last performed sensor reading $\hat{v}$ prior to t and the actual sensor

Algorithm 2: ARTC.next

Parameters: setpoint E, maximal interval diameter $dMax$
State : shift $s=0$, diameter $\Delta=0$, last value v_l
Input: read-time t, value v, desired read-time t_D
Output: next interval borders $[t_s,t_e]$,

```
μ_v = pewma.next(‖v‖_2);
if v_l is set then
    μ_δ, σ_δ = pewmaDiff.next(‖v−v_l‖_2);
    s += pid.next((μ_δ/μ_v)) · 0.1;
    δ_E = σ_δ/(μ_v·E) − 1;
    if (δ_E > 0) Δ ·= 1/2 else Δ −= δ_E;
    ensure s ∈ [−1/2, 1/2] and Δ ∈ [0,dMax];
else
    Δ = 0;
end
t_e, t_s = t_D + Δ · (s ± 1/2);
v_l = v;
return t_s, t_e;
```

value v_i at time t. We label ϵ the average error over the entire experiment trace, and denote the error's moving average with window size k in percent of the data's average magnitude via ϵ^k.

During all conducted experiments, we compare the induced error ϵ of ARTC to the error of fixed read-time tolerance algorithms that we configured to achieve the same average read-time tolerance μ_Δ as ARTC. We use the read-time suggestion algorithm AdaM throughout the experiments, and only vary the parameter E of ARTC and the fixed read-time tolerance algorithm accordingly. The parameters that are fixed throughout the experiments are provided in Table 2.

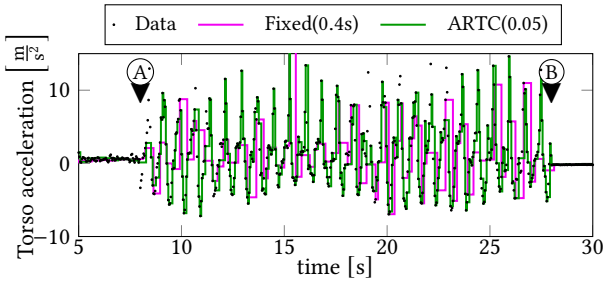
3.3 Experimental Evaluation

We first present the evaluation on an experiment trace on an excerpt from the activities dataset and then provide quantitative results for all three datasets.

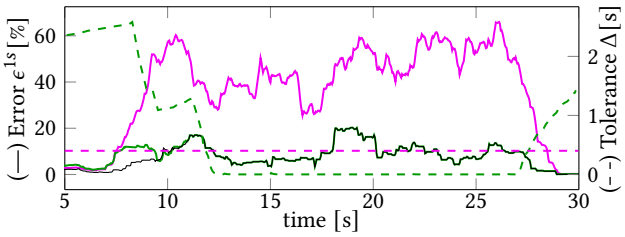
Experiment Trace. Figure 2 shows the performance of ARTC through an experiment trace excerpt from the activities dataset. We execute a total of three query-configurations on the dataset (i) A baseline query without read-sharing, which executes only the read-time suggestion algorithm AdaM, (ii) ARTC configured with $E = 0.05$, and (iii) A fixed read-time tolerance algorithm configured with $\Delta = 0.4$ seconds (s).

In Figure 2a, we visualize the raw sensor data at the highest possible rate as black dots. The representation of the sensor provided through configurations (ii) and (iii) is visualized as solid lines, which indicate the last performed sensor reading $\hat{v}_i$ at any given point in time. At any point in time, the difference between the solid lines and the data points is the error induced by the respective query. Figure 2b visualizes the moving average of the relative error over one second and the read-time tolerance Δ_i.

During the visualized experiment trace, a person performs three operations: standing in an elevator still until Ⓐ, exercising on a stepper until Ⓑ, and sitting. Every time the performed activity changes, we observe a dramatic change in the distribution of the replayed data. While exercising, the variability of the replayed torso acceleration measurements increases significantly.
At the beginning of the experiment, the variability of the data is relatively low. Both the fixed read-time tolerance algorithm and



(a) The black dots show acceleration measurements at the highest available rate. Solid lines visualize the last sensor reading performed with ARTC resp. the fixed read-time tolerance algorithm.



(b) Diameter Δ and 1-second moving average of the error ϵ induced by ARTC, the fixed read-time tolerance algorithm, and without read-sharing (black) altogether. During volatile phases, ARTC decreases Δ and thus reduces ϵ to the error without read-sharing.

Figure 2: Experiment trace of ARTC and baselines.

ARTC achieve a similar error ϵ of less than 5%. However, during this low variability phase, ARTC enables a read-time tolerance larger than 2.4s, which is more than six times larger than those achieved by the fixed read-time tolerance algorithm.
A brief spike in the torso acceleration precedes the individual's exercising routine at the 5-second mark, which leads to ARTC reducing the read-time tolerance. Once the torso acceleration starts to oscillate (Ⓐ), ARTC reduces the read-time tolerance to a minimum, such that the error levels with those induced by the read-time tolerance algorithm AdaM. During this phase, the read-time tolerance proposed by the fixed algorithm is higher than those proposed by ARTC at the cost of an error ϵ of approximately 50% of the data's magnitude, while ARTC causes no additional error. Once the individual sits (Ⓑ), ARTC increases the read-time tolerance again.

By controlling the read-time tolerance based on the data's variability, ARTC is able to outperform the fixed read-time tolerance algorithm in both depicted performance metrics: When considering the entirety of the depicted trace, the read-time tolerance of the fixed read-time tolerance algorithm is approximately doubled by ARTC, while ARTC can reduce the average error by 75%.

Quantitative Experiments. To assess the performance of ARTC under different circumstances, we run multiple experiments on the gas, activities, and football datasets. We compare ARTC to fixed read-time tolerance algorithms achieving the same average read-time tolerance Δ in Figure 3. Throughout the experiments, we observe that ARTC outperforms or at least levels the fixed read-time tolerance algorithm. Overall, ARTC outperforms the fixed read-time tolerance algorithm by the largest margin on the activities dataset: on average, it reduces the error by 31% compared to approximately 15% on the other two datasets. However, the additional error induced by ARTC for enabling read-sharing is smallest on the gas dataset (1.3 percentage points for $\Delta > 4$s). This relatively small induced error reflects the relatively steady nature

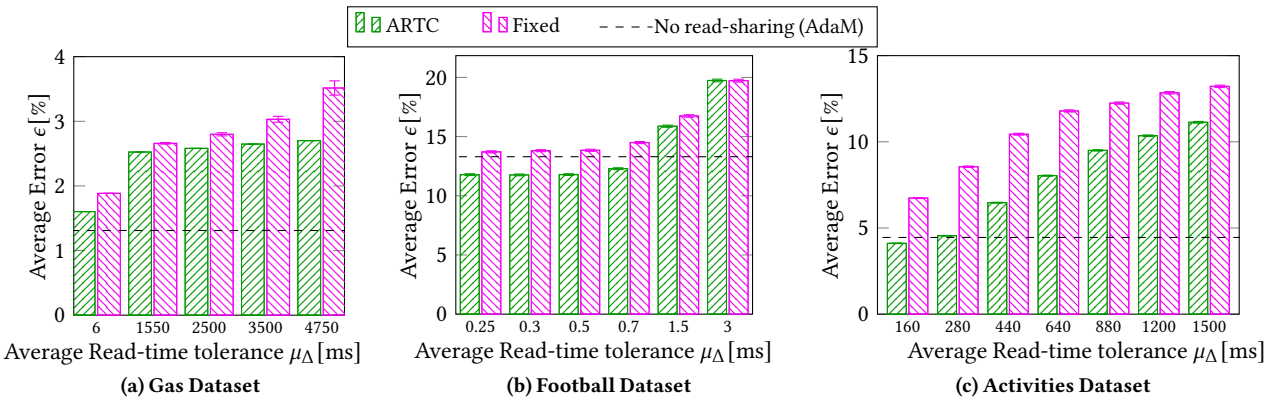


Figure 3: Performance of queries with different configurations of ARTC and the two baselines on the introduced datasets.

of the gas data. We discuss the experimental results in detail for each dataset in the following paragraphs.

The gas dataset (Figure 3a) contains sensor values with the lowest variability of all three datasets. The error of the read-time suggestion algorithm AdaM without read-sharing is approximately 1.4% of the data's average magnitude. With the configurations of ARTC visualized in Figure 3, large read-time tolerances of up to 4.75s are possible.
While small deviations on the dataset can only be captured when enabling read-sharing only to a limited extent of 6ms, we observe that ARTC is able to allow for large read-time tolerances of 2−5s without a significant increase in the average error. On the other hand, the fixed read-time tolerance algorithm's performance deteriorates, such that the error of the fixed algorithm is 30% larger than those caused by ARTC for a read-time tolerance of 4.75s. The additional error of read-sharing through ARTC stays below the error induced by the read-time suggestion algorithm for all executed configurations.

The football dataset (Figure 3b) has the highest variability of all presented datasets. The error of the read-time suggestion algorithm alone is 13% of the data's average magnitude. As most of the configurations of ARTC indicate an average error below or at the same level as the error of the read-time suggestion algorithm, read-sharing is only enabled to a very limited extent, up to 3ms. On this dataset, we observe that ARTC is able to reduce the error of the read-time tolerance algorithm slightly by up to 13% when enabling read-sharing to a limited extent. For read-time tolerances of 1.5ms, the error of the query configured with ARTC (15.8%) draws nearer to those of the fixed algorithm (16.7%). For a read-time tolerance of 3ms, the performance of both algorithms levels.

The read-time suggestion algorithm on the activities dataset (Figure 3c) has an error of 4.4%, matched for ARTC up to a read-time tolerance of 280ms. The executed configurations of ARTC achieve a read-time tolerance of up to 1.5s. On this dataset, ARTC outperforms the fixed read-time suggestion algorithm by the largest margin. ARTC is able to adapt to changes in the variability of the data during the experiment that occur whenever the individuum starts performing a different activity. While the error of the fixed read-time tolerance algorithm is approximately 90% larger than those of ARTC for a read-time tolerance of 280ms, the margin between the performance of both algorithms continuously shrinks to 20% for a read-time tolerance of 1.5s. This is because the algorithm misses small fluctuations in the data and takes a long time to adapt to brief periods of activity, which only last for several seconds.

3.4 Discussion

The evaluation of the single trace experiment executed on the activities dataset shows that ARTC can quickly adapt to a change in the distribution of values gathered from the sensor. During the quantitative analysis conducted on the three different datasets, ARTC outperforms the corresponding fixed configuration. For our experiment configurations, it is even possible to enable read sharing to a limited extent without a detrimental effect on the accuracy of the data representation altogether.

4 RELATED WORK

To the best of our knowledge, we are the first to propose a general, adaptive demand-based read-time tolerance controller. However, the underlying problem of reducing the number of reads and transmissions has been studied from various angles. Therefore, we first present existing work on read- and transmission sharing. We then present adaptive sampling techniques, as they are an integral building block of ARTC and share a similar objective.

Read- and transmission sharing. TinyDB [12] introduces the concept of acquisitional query processing (ACQP) to control the sampling frequency during *Single-Query Optimization*. ACQP arranges database operators and sensor readings in a common processing pipeline. Operators with low selectivity reduce the acquisition frequency by filtering out sensor readings before succeeding read operations. However, TinyDB only allows for periodic sampling algorithms at the processing pipeline's source.

Multiple approaches known from the literature conduct *spatial resource sharing*, i.e., optimize the set of deployed queries through a global view into a single query. Li et al. [10] allow the user to define query priorities and -deadlines. They fuse aggregate queries accordingly, perform utility-driven compression, and global transmission-scheduling to save reads and transmissions. However, they do not adapt to the variability of the distribution in scheduling and fusing sensor readings.

As opposed to spatial resource sharing, *data sharing techniques* perform local optimizations using a single sensor reading for multiple queries [13, 24, 25]. Tavakoli et al. [18] model the overlaps of tolerance intervals in an online evolving interval-cover graph, which they use to determine read-times. All four approaches are unable to adapt to changes in the distribution of the data both in scheduling reads and in computing read-time tolerances. We advanced data sharing in the context of adaptive sampling techniques [19, 20]. To that end, we proposed to combine demand-based adaptive read-time suggestion and read-time fusion in a

single framework. We were able to report savings of 87% in reads and transmissions, which we further increase through ARTC.

Adaptive sampling tailors the sampling frequency to the distribution of the data [8]. Padhy et al. propose a confidence-based adaptive sampling method called Utility-based Sensing and Communication (USAC) [15]. They apply linear regression to predict the next sensor value with a bounded error-range, the so-called confidence interval (CI). If a value is outside the CI, the sensor starts sampling at maximal frequency. Otherwise, the frequency decreases exponentially by a factor $\alpha \in [0,1]$ until it reaches the minimum sampling frequency.

Aiming to provide an energy-efficient solution in the realm of Big Data and IoT, Trihinas et al. propose the Adaptive Monitoring Framework (AdaM) [23]. They use an ad-hoc forecasting method called PEWMA to produce one-step forecasts, which they then use to compute the metric stream's variability.

Fan et al. propose Filtering and Adaptive sampling for Differentially Private Time Series Monitoring (FAST) [6]. They define a so-called privacy budget to add Laplace noise to the original observations to achieve differential privacy. Then they generate estimates, the quality of which is then used by the sampling component to adjust the sampling rate using a PID controller. Compared to AdaM and USAC, FAST is slower to adapt to changes in the distribution of the data but achieves comparable results.

We design our adaptive read-time tolerance controller ARTC using ideas from all three of the aforementioned adaptive sampling algorithms. We use a PID controller [3] in order to assess whether the read-time suggestion algorithm achieves the data-quality ARTC targets. Similar to AdaM, we use PEWMA [5] in order to monitor the distribution of samples. We use the idea presented by Padhy et al. of decreasing tolerances rapidly if the desired data accuracy is missed, which enables us to adapt to changes in the distribution quickly.

5 CONCLUSION

We previously developed a multi-query read-scheduling algorithm that enables adaptive sampling in a sensor network [20]. In this paper, we now extend our work by proposing the easy to configure adaptive read-time tolerance controller ARTC. Our experimental evaluation shows that ARTC tailors the extent of read-sharing to the data-accuracy demands of end-applications. ARTC is generally-applicable for defining read-time tolerances when scheduling sensor read-times. Thus, it enables multi-query optimization through sharing sensor readings for arbitrary adaptive sampling techniques. We evaluate ARTC on three real-world IoT datasets with different data characteristics and shifts in the data distribution. Our solution reduces the error in the representation by up to 60% compared to fixed read-time tolerance algorithms by adapting to the sensed data's variability. ARTC not only reduces the number of reads and transmissions while achieving the same sensing error as existing techniques, but also enables multi-query read-sharing for queries issued by users without domain-knowledge. We make our code and evaluation available open-source. We also provide detailed instructions on how to execute custom experiments. In our previous work, we allow the user to define a so-called penalty-function [20] to further increase the read-sharing potential under specific circumstances. We plan to extend our solution to adaptively tune such penalty-functions based on the data's variability as well.

Acknowledgements: This work has been supported by German Ministry for Education and Research as BIFOLD (01IS18025A, 01IS18037A).

REFERENCES

[1] Kerem Altun and Billur Barshan. 2010. Human activity recognition using inertial/magnetic sensor units. In *International workshop on human behavior understanding*. Springer, 38–51.

[2] M Araki. 2009. PID control. *Control Systems, Robotics and Automation: System Analysis and Control: Classical Approaches II, Unbehauen, H.(Ed.). EOLSS Publishers Co. Ltd., Oxford, UK., ISBN-13: 9781848265912* (2009), 58–79.

[3] Karl J Åström and Tore Hägglund. 2006. Advanced PID control. In *The Instrumentation, Systems, and Automation Society*. Citeseer.

[4] A Zul Azfar and Desa Hazry. 2011. A simple approach on implementing imu sensor fusion in pid controller for stabilizing quadrotor flight control. In *7th International Colloquium on Signal Processing and its Applications*. IEEE, 28–32.

[5] Kevin M Carter and William W Streilein. 2012. Probabilistic reasoning for streaming anomaly detection. In *Statistical Signal Processing Workshop (SSP)*. IEEE, 377–380.

[6] Liyue Fan, Li Xiong, and Vaidy Sunderam. 2013. FAST: differentially private real-time aggregate monitor with filtering and adaptive sampling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1065–1068.

[7] Jordi Fonollosa, Sadique Sheik, Ramon Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.

[8] Dimitrios Giouroukis, Alexander Dadiani, Jonas Traub, Steffen Zeuch, and Volker Markl. 2020. A Survey of Adaptive Sampling and Filtering Algorithms for the Internet of Things. In *DEBS'20: 14th ACM International Conference on Distributed and Event-Based Systems*.

[9] Julius Hülsmann, Jonas Traub, and Volker Markl. 2020. Demand-based Sensor Data Gathering with Multi-Query Optimization. In *Proceedings of the 46th Conference on Very Large Databases*, Vol. 13. VLDB Endowment. Issue 12.

[10] Ming Li, Tingxin Yan, Deepak Ganesan, Eric Lyons, Prashant Shenoy, Arun Venkataramani, and Michael Zink. 2007. Multi-user data sharing in radar sensor networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems*. ACM, 247–260.

[11] Chong Liu, Kui Wu, and Jian Pei. 2007. An energy-efficient data collection framework for wireless sensor networks by exploiting spatiotemporal correlation. *IEEE transactions on parallel and distributed systems* 18, 7 (2007), 1010–1023.

[12] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)* 30, 1 (2005), 122–173.

[13] Rene Muller and Gustavo Alonso. 2006. Efficient sharing of sensor networks. In *International Conference on Mobile Ad Hoc and Sensor Systems*. IEEE, 109–118.

[14] Christopher Mutschler, Holger Ziekow, and Zbigniew Jerzak. 2013. The DEBS 2013 grand challenge. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 289–294.

[15] Paritosh Padhy, Rajdeep K Dash, Kirk Martinez, and Nicholas R Jennings. 2006. A utility-based sensing and communication model for a glacial sensor network. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. ACM, 1353–1360.

[16] Chana R Schoenberger. 2002. The internet of things. *Forbes* (2002), 155160–155160.

[17] Amit Kumar Sikder, Giuseppe Petracca, Hidayet Aksu, Trent Jaeger, and A Selcuk Uluagac. 2018. A Survey on Sensor-based Threats to Internet-of-Things (IoT) Devices and Applications. *arXiv preprint arXiv:1802.02041* (2018).

[18] Arsalan Tavakoli, Aman Kansal, and Suman Nath. 2010. On-line sensing task optimization for shared sensors. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 47–57.

[19] Jonas Traub. 2019. *Demand-based data stream gathering, processing, and transmission*. Ph.D. Dissertation. Technische Universität Berlin.

[20] Jonas Traub, Sebastian Breß, Tilmann Rabl, Asterios Katsifodimos, and Volker Markl. 2017. Optimized On-demand Data Streaming from Sensor Nodes. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 586–597. https://doi.org/10.1145/3127479.3131621

[21] Jonas Traub, Julius Hülsmann, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. SENSE: Scalable Data Acquisition from Distributed Sensors with Guaranteed Time Coherence. *arXiv preprint arXiv:1912.04648* (2019).

[22] Demetris Trihinas, Luis F Chiroque, George Pallis, Antonio Fernández Anta, and Marios D Dikaiakos. 2018. ATMoN: Adapting the" Temporality" in Large-Scale Dynamic Networks. In *2018 IEEE 38th International Conference on Distributed Computing Systems*. IEEE, 400–410.

[23] Demetris Trihinas, George Pallis, and Marios D Dikaiakos. 2015. AdaM: An adaptive monitoring framework for sampling and filtering on IoT devices. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 717–726.

[24] Shili Xiang, Hock Beng Lim, and Kian-Lee Tan. 2006. Impact of multi-query optimization in sensor networks. In *Proceedings of the 3rd workshop on Data management for sensor networks: in conjunction with VLDB 2006*. ACM, 7–12.

[25] Shili Xiang, Hock Beng Lim, Kian-Lee Tan, and Yongluan Zhou. 2007. Two-tier multiple query optimization for sensor networks. In *27th International Conference on Distributed Computing Systems (ICDCS'07)*. IEEE, 39.

[26] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform: Data and application management for the internet of things. *CIDR* (2020).

Short Paper

SOJA: A Memory-efficent Small–large Outer Join for MPI

Liang Liang, Guang Yang, Thomas Heinis
Imperial College London

David Taniar
Monash University

ABSTRACT

The join is a fundamental and widely used operation in data analytics but equally, it is also one of the most expensive ones. Considerable work has been carried out to improve and evaluate join approaches based on popular distributed processing systems such as Spark and Hadoop, however, it has not been widely studied on MPI.

In this paper, we first implement, analyse and compare existing algorithms for the common small-large outer join operation and develop a novel approach, the swap-based outer join algorithm (SOJA). SOJA is designed to minimise the expensive communication between the distributed nodes while also reducing the cost of the local join operations. We demonstrate the benefits of SOJA experimentally, showing that it achieves at worst an execution time similar to its competitors. More importantly, SOJA requires substantially less memory (typically half the memory compared to the best competitor) and that memory usage scales very well.

KEYWORDS

Outer Joins, Algorithm, Parallel Processing, MPI

1 INTRODUCTION

Collecting and storing data has never been as easy and as cheap as today. It comes as no surprise that vast amounts of data are being stored today and it is predicted that all known data worldwide will grow to 250 Zettabytes by 2025. Many real-world applications benefit from the efficient analysis of large amounts of data, be it for medical applications [1, 2, 13], scientific applications [11] or commercial applications such as traffic analysis [7] and others. Analysing this data efficiently and at scale has thus never been more important than today and is also a considerable challenge.

Crucial in the analysis of large amounts of data is the combination of multiple datasets before analysis. One central operation thus is the join operation which combines multiple datasets (or one with itself) by matching tuples with a shared attribute. More specifically, a join on datasets R and S based on equality (or a different relationship) will pair tuples $r \in R$ and $s \in S$ if $r.c_1 = s.c_2$ where c_1 and c_2 are attributes of the tuples. The operation is frequently used but very costly due to computational overhead but also because of I/O.

In this paper, we develop the swap-based outer join algorithm (SOJA), a new approach to the specific problem of the small-large outer join where a small and a large dataset are joined. We develop SOJA for Message Passing Interface (MPI) on HPC infrastructure as such large-scale parallel infrastructure is one of the few efficient ways to join large datasets [3]. MPI is a message-passing standard which is widely used in high performance applications [10]. The standard defines the syntax and semantics of approximately 250 library routines that allow users to develop a wide variety of communication operations on different types of parallel computing infrastructure[8, 9]. Point to point communication between two MPI processes (ranks) and collective communication among MPI processes are most commonly used. For each of them, MPI also provides multiple communication modes that fall into either blocking or non-blocking communication according to whether constituent operations of the communication complete synchronously. Additionally, MPI supports a derived datatype and a virtual topology which allows users to control data movement among processes efficiently and flexibly. MPI thus is a promising tool to design and implement algorithms to handle and analyze massive amounts of data efficiently.

We use MPI running on HPC infrastructure to efficiently execute a small-large left outer join. The small-large left outer join can be denoted as $R \bowtie S$ with $|R| << |S|$ where $|R|$ and $|S|$ are number of tuples in tables or, more generally, datasets R and S. In the query below, R and S represent the left table and right table, and a join is performed between R and S based on join keys $R.a$ and $S.b$:

```
SELECT * FROM R LEFT OUTER JOIN S ON R.a = S.b;
```

For parallelising the left outer join, we assume that R and S are distributed among the N processes in a round-robin fashion. Each partition R_i is assigned to the $i-th$ process, and has the same number of elements $|R_i| = \frac{|R|}{N}$. The same applies to all S_i.

The parallel left outer join with partitioned data then has two goals: (1) find all matching tuples from the two tables; (2) find all dangling tuples from the left table and output them with no matching tuple from the right table.

Existing approaches mainly adopt two methods, redistribution and broadcast, to produce the entire join results while guaranteeing data locality. Redistribution refers to redistributing both tables among all processes to make tuples such that the same join keys are placed in the same process. Broadcast, in this context, means the left table in each process is duplicated and sent to all other processes, so that each process holds the complete left table. These two methods either lead to inevitable skewness or duplication. SOJA adopts a novel method, *swap*, to ensures tuples in the left table can join all possible matching tuples in the right table by swapping the left table among processes. Based on the swap method, SOJA can also perform other types of parallel joins such as inner or right outer joins by replacing local join methods.

As our extensive set of experiments shows, SOJA in many cases outperforms its competitors and at worst has an execution time similar to its competitors. Most importantly, however, SOJA requires substantially less memory which in a supercomputing environment is crucial (as data cannot easily be swapped to disk). SOJA typically requires only half the memory and, as our experiments show, scales extremely well.

In the remainder of this paper we first review the state of the art in Section 2, present our approach SOJA in Section 3.1, analyse SOJA experimentally in Section 4 and conclude in Section 5.

2 RELATED WORK

In this section, we first describe four related parallel outer join algorithms (Figure 1) and then discuss their limitations and communication implementations on MPI .

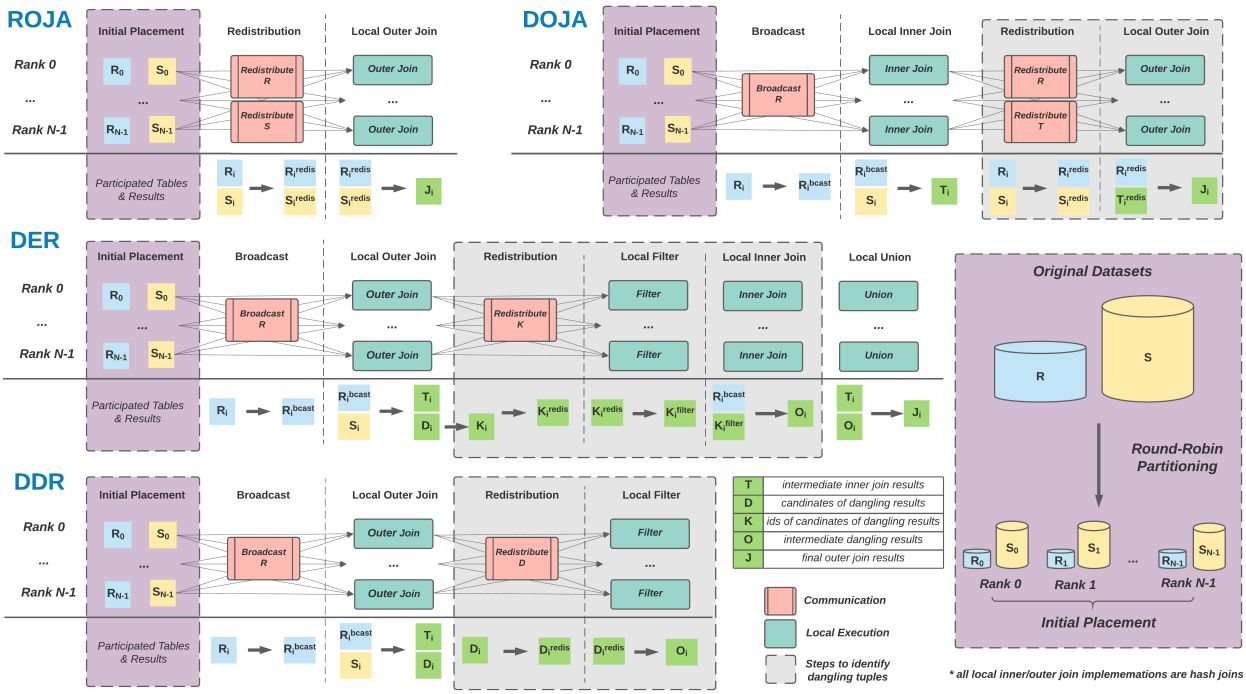


Figure 1: Illustration of ROJA, DOJA, DER and DDR

2.1 Redistribution Outer Join Algorithm (ROJA)

ROJA uses the redistribution approach to execute the local outer join, it has two steps, redistribution and local outer join, which can be considered as an extension of a typical re-partition join algorithm [4, 12]. The only difference to a typical re-partition join algorithm is that ROJA adopts the local outer join in the second step instead of local inner join. ROJA proceeds as follows:

(1) all tuples of R_i and S_i in each process are redistributed based on the join keys ($R_i.a$ and $S_i.b$).
(2) each process performs a local outer join between reallocated two tables denoted as R_i^{redis} and S_i^{redis}.

2.2 Duplication Outer Join Algorithm (DOJA)

DOJA is a variation of the broadcast join algorithm [4, 12]. It obtains inner join results by performing inner join after broadcasting the tables similar to broadcast join algorithms. However, DOJA needs to execute two additional steps to identify dangling tuples:

(1) each process broadcasts the smaller left table R_i to all other processes, which results in each process now having an entire table R denoted as R_i^{bcast}.
(2) local inner join in each process between R_i^{bcast} and S_i outputs inner join results stored in T_i.
(3) T_i and R_i are redistributed based on join keys ($T_i.a$ and $R_i.a$).
(4) after receiving redistributed T_i^{redis} and R_i^{redis}, each process executes local outer join to output complete result.

2.3 Duplication and Efficient Redistribution (DER)

DER [14] improves DOJA by reducing the size of the redistributed data. DER executes as follows:

(1) this step is same as step 1 for DOJA with the only difference being the addition of ids. DDR needs to add keys to tuples, if there is no global identifier in R_i.
(2) a local outer join between R_i^{bcast} and S_i generates inner join results (T_i) as well as non-matching tuples.
(3) each process in DER redistributes ids (K_i) extracted from non-matching tuples (D_i).
(4) after redistribution, K_i^{redis} is filtered so that only keys that appear N times are stored, because a dangling tuple in R will not be matched in all N processes. The result is in K_i^{filter}.
(5) truly dangling tuples O_i in the left table can be retrieved by a local inner join between K_i^{filter} and R_i^{bcast}.
(6) the union of O_i and T_i in each process is the final results.

2.4 Duplication and Direct Redistribution (DDR)

DDR [5] is a broadcast-based outer join algorithm focusing on optimizing the identification of dangling tuples. Its steps are:

(1) the first two steps of DDR are identical to DER.
(2) each process outputs matched join results (T_i) and redistributes non-matching results (D_i).
(3) similar to step 4 of DER, dangling tuples after redistribution (D_i^{redis}) are kept if they occur N times. The results O^i is directly outputted.

2.5 Discussion

We will compare the communication cost and local execution cost between the different approaches.

2.5.1 Communication. The two main methods for communicating among processes are: (1) redistributing where the tuples are categorized and sent to their respective destinations; (2) broadcasting, where a table is duplicated across all of the

processes. Redistribution requires additional calculation of destination, thus, the redistribution cost is higher than broadcast cost ($t^{redis} > t^{trans}$). To compare communication cost of algorithms, we develop a simple cost model.

$$comm \begin{cases} (R+S) \times t^{redis} & \text{(ROJA)} \\ R \times (N-1) \times t^{trans} + (R+T) \times t^{redis} & \text{(DOJA)} \\ R \times (N-1) \times t^{trans} + K \times t^{redis} & \text{(DER)} \\ R \times (N-1) \times t^{trans} + D \times t^{redis} & \text{(DDR)} \end{cases} \quad (1)$$

In small-large joins, based on the above equation, the communication cost of ROJA can be very high, since each process in ROJA has to redistribute the large right table S_i. The other three algorithms share a similar broadcast cost. The cost may appear low initially, but with an increase in the number of processes N and the size of left table ($|R|$), the cost can grow significantly. The difference in communication cost between the three broadcast-based algorithms is that they redistribute different intermediate data. DOJA redistributes the left table R and the inner join result T (the size of T depends on the selectivity ratio σ_j). With an increasing σ_j, the redistribution cost will increase significantly, thereby dominating the overall performance of DOJA. As a consequence, two improved methods, DDR and DER, were developed to optimize this stage by redistributing dangling results. DER uses ids of dangling tuples (K) and DDR uses dangling tuples (D). The number of dangling ids ($|K|$) is the same as the number of dangling tuples ($|D|$), however, the storage size of K may be smaller than D. Thus, from the perspective of communication, DER can outperform DDR. The overall redistribution cost of DDR and DER is not significant. However, the number of dangling tuples will increase when the number of processes (N) increase simply due to the fact that the R is duplicated in more processes. It is worth mentioning that redistributing may potentially result in an unbalanced workload due to skewness. In the presence of data skew, the overall performance is adversely impacted by the redistribution stage as well as subsequent operations. Additionally, both redistribution and broadcast require more memory.

2.5.2 Local Execution. The existing algorithm mainly involve two local operations: join (inner join and outer join) and filter. Their time costs are denoted as t^{join} and t^{filter}.

$$local \begin{cases} (R_i^{redis} + S_i^{redis}) \times t^{join} & \text{(ROJA)} \\ (R + S_i + R_i^{redis} + T_i^{redis}) \times t^{join} & \text{(DOJA)} \\ (2R + S_i + K_i^{filter}) \times t^{join} + K_i^{redis} \times t^{filter} & \text{(DER)} \\ (R + S_i) \times t^{join} + D_i^{redis} \times t^{filter} & \text{(DDR)} \end{cases} \quad (2)$$

As we mentioned before, the skewness in the data can result in an unbalanced workload in subsequent operations and the performance of ROJA is highly affected by data skew (as data in some processes after redistribution (R_i^{redis}, S_i^{redis}) may be significantly bigger than the initial placement (S_i , R_i)). DOJA not only needs to perform a local join after the broadcast to compute all inner join results but also needs to identify non-matching tuples in the left table by joining the redistributed left table (R_i^{redis}) and inner join results (T_i^{redis}). The cost of a second join strongly depends on the inner join ratio (σ_j) and skewness degree (θ). For DER and DER, a filter step is required after the local join. DER needs to filter keys which may have a smaller size in bytes than the tuples filtered by DDR. Thus, in terms of cost of the filter stage, DER has an advantage. However, after the filter stage, DDR can directly output dangling tuples while DER has to retrieve dangling tuples by performing an additional join. In addition, DER has extra local update costs ($R_i \times t^{update}$) if no ids are assigned to the left table R_i.

2.6 Broadcast and Redistribution on MPI

Both broadcast and redistribution are collective communication in which all processes are participating. For broadcasting in DOJA, DER and DDR, all processes need to send their participating partitions to all other processes. From the point of view of the receiver, all processes gather data from all other processes, which can be achieved by the MPI routine `MPI_Allgather`. Different processes may hold data of different size, we implement the broadcast by using `MPI_Allgatherv` which allows each process to contribute different amounts of data. As for redistribution, all processes could be the destination of the redistributed data in all processes. Such a communication pattern can be supported by `MPI_Alltoall` and `MPI_Alltoallv` which send data from all processes to all processes. To implement hash redistribution, before calling these routines, the data needs to be reorganized based on its corresponding destinations which is determined by the hash function.

3 SWAP-BASED OUTER JOIN ALGORITHM (SOJA)

3.1 Approach & Design of SOJA

Compared to existing algorithms that mainly uses redistribution and broadcasting, SOJA adopts a novel *swap* approach using a ring topology to perform the parallel outer join. Additionally, SOJA decomposes the traditional local outer hash join operations to hash (create hash table from right table) and lookup (loop over tuples in R_i to look up the hash table). Hashing the right table allows the dangling tuples in the left table to be determined with a single loop for a left outer join. The steps for SOJA are:

(1) each process creates a hash table (HS_i) for the right table (S_i). It then loops over the tuples in the left table (R_i) to query the hash table HS_i, meanwhile marking non-matching tuples in the R_i as candidates for dangling tuples. Finally, the intermediate inner join results will be directly returned.
(2) each process sends the updated R_i to the next process ($i+1$) in the ring topology. Note, the $(N-1)-th$ process will send R_{N-1} to process 0.
(3) a lookup would be carried out on the new set of R_i and HS_i, and the intermediate results would be outputted. The non-matching mark will be removed if the candidate tuple is able to find a match.
(4) repeat steps 2 & 3 $N-1$ times, and output the tuples in each of the R_i with marks as dangling tuples in the last iteration.

The cost model of the iterative operations is the following:

$$SOJA \begin{cases} R \times (N-1) \times t^{trans} & \text{(comm)} \\ (R + S_i) \times t^{join} + R \times t^{update} & \text{(local)} \end{cases} \quad (3)$$

From perspective of cost model, the communication cost is the same as the broadcast operation, the total join cost of SOJA is also the same as outer join after the broadcast in DER and DDR. However, no redistribution, filter, or additional joins are required by SOJA. It identifies non-matching tuples in the left table and updates the left table R while looping. Additionally, the cost of t^{update} is not significant, due to the left table being small.

Further, compared to existing methods, SOJA saves memory for the following reasons: (1) directly outputs of the intermediate

results after each iteration, (2) no duplication of tables, and (3) no temporary dangling data as the size of temporary data can increase with the number of processes as well as with the selectivity ratio in the left table σ_R. In addition, SOJA can easily be extended to perform other types of joins such as inner join or right outer joins. For example, inner join is executed by performing a local lookup without marking and updating dangling tuples in the left table. For right outer joins, SOJA simply flips the operation by creating a hash table for the left table and updating the marks on the right table.

Although, there is no redistribution in SOJA, its performance is still sensitive to an imbalanced workload among processes, especially due to imbalanced initial placement as there are synchronization costs for communication. Therefore, for SOJA, initially placing partitions of equal size is a prerequisite for achieving ideal performance.

3.2 Swap of SOJA on MPI

We use Cartesian topology routines `MPI_Cart_create` with 1-dimension to construct SOJA's ring topology in which each process is logically connected to two other processes ($left$ and $right$ process) in a circle. In the ring topology, point to point communication routines (`MPI_Send` and `MPI_Recv`) can be used to swap data between adjacent processes. To reduce the synchronization costs, we use non-blocking mode (`MPI_Isend` and `MPI_Irecv`) to transfer data that can overlap other local operations, such as result output or other operations based on intermediate results. The swap stage of SOJA is shown in Algorithm 1.

Algorithm 1: Swap in SOJA

```
1:  Require: R_i, N, left and right neighbour left and right
2:  recv_buffer = R_i
3:  for i = 1 to i = N-1 do
4:      send_buffer = lookup_and_update(recv_buffer)
5:      MPI_Isend(send_buffer, dest = left, send_Request)
6:      MPI_Irecv(recv_buffer, source = right, recv_Request)
7:      overlapping: output result
8:      MPI_Wait(send_Request)
9:      MPI_Wait(recv_Request)
10: end for
11: result = lookup_and_update(recv_buffer)
```

4 EXPERIMENTAL EVALUATION

In the following section, we analyse SOJA using experiments with changing the size of left table ($|R|$), selectivity ratio of left table (σ_R), and skewness degree (θ). We focus on execution time and memory usage.

4.1 Setup

4.1.1 Platform. We use an HPC cluster with Intel E5-2680 v3 @ 2.50GHz running Centos 8. The MPI version is 3.3.2.

4.1.2 Datasets. The experiments are based on `customer` and `supplier` tables from the TPC-H benchmark [6]. For simplicity and consistency of presentation, we use R and S to represent `customer` and `supplier` respectively. To test the algorithms with different scenarios, we modify the data in two ways: first, we scale up or scale down the table size ($|R|$ and $|S|$) by sampling data from the two initial tables (uniform with replacement). Second, we vary the selectivity ratios, i.e., the join selectivity ratio (σ_j) and the selectivity ratio in the left table (σ_R). The selectivity ratio affects the number of join results ($|T|$) and can be roughly controlled by the sample population of join key n with uniform distribution, which can be explained using Equations 4 to 6.

The definition of σ_j is:

$$\sigma_j = \frac{|T|}{|R| \times |S|} \tag{4}$$

If we sample the join key from a population that contains n values and denote the probability of choosing $i-th$ item as p_i, then:

$$\sigma_j = \frac{\sum_{i=1}^{n} p_i \times |R| \times p_i \times |S|}{|R| \times |S|} = \sum_{i=1}^{n} p_i^2 \tag{5}$$

If p_i is a uniform distribution, $p_i = \frac{1}{n}$, then:

$$\sigma_j = \frac{1}{n} \tag{6}$$

The selectivity ratio in the left table (σ_R) determines the number of dangling tuples in the left table. With the involvement of σ_R, the number of results can be estimated by Equation 7.

$$|T| = \underbrace{|R| \times \sigma_R \times |S| \times \sigma_j}_{\textit{inner join results}} + \underbrace{(1-\sigma_R) \times |R|}_{\textit{dangling results}} \tag{7}$$

If σ_R equals to 0, this means no tuples in table R are selected to perform the join with table S. Therefore, table R will be the result. As σ_R increases, the number of matching results increases whereas the non-matching results decrease until all tuples in R are involved in the join when σ_R equals to 1. We assign a negative join key to number of tuples in table R based on 1 - σ_R [5]. Therefore, any tuple with a negative key in the left table (R) will not have any matching results. Both σ_j and σ_R can influence $|T|$, but in outer joins, σ_R will provide more insights (dangling tuples), so our experiments focus on σ_R to study changes of $|T|$ and only use σ_j to keep the number of potential inner join results (when σ_R = 1) unchanged when varying the input data size.

To achieve the parallel IO, we split the data into N partitions and processes load their partition from disk in parallel.

4.2 Impact of the Size of the Left Table

To examine how the performance of approaches changes when the size of left table increases, we conduct the experiment in which we vary $|R|$ and keep all other parameters such as $|S|$, σ_R, fixed. All five algorithms are executed on 32 cores and 64 cores and we measure execution time and memory usage.

In this experiment, we set $|S|$ to 5×10^7, σ_R to 0.5, and increase $|R|$ with a coarser granularity (10×) from 10^3 to 10^7. Figure 2 shows that all algorithms maintain a stable performance until $|R|$ reaches 10^6. We further explored changes of the performance with a fine-grained increase of $|R|$ (increment of 10^6 tuples) in the interval between 10^6 and 10^7. The result from 32 processes and 64 processes (Figure 2 A & C) shows that the execution time of ROJA is steady whereas execution time of all other algorithms has a clear upward trend. Figures 2 B & D indicate that SOJA has an advantage in terms of memory usage across all experiments. Although ROJA has the highest memory usage when $|R|$ is small, the memory usage level stays quite constant with increasing $|R|$. The other three broadcast-based algorithms show that their memory usage increases significantly, particularly DDR. It is worth pointing out that when $|R|$ is relatively large, the performance of DDR on 64 processes is worse than itself on 32 processes.

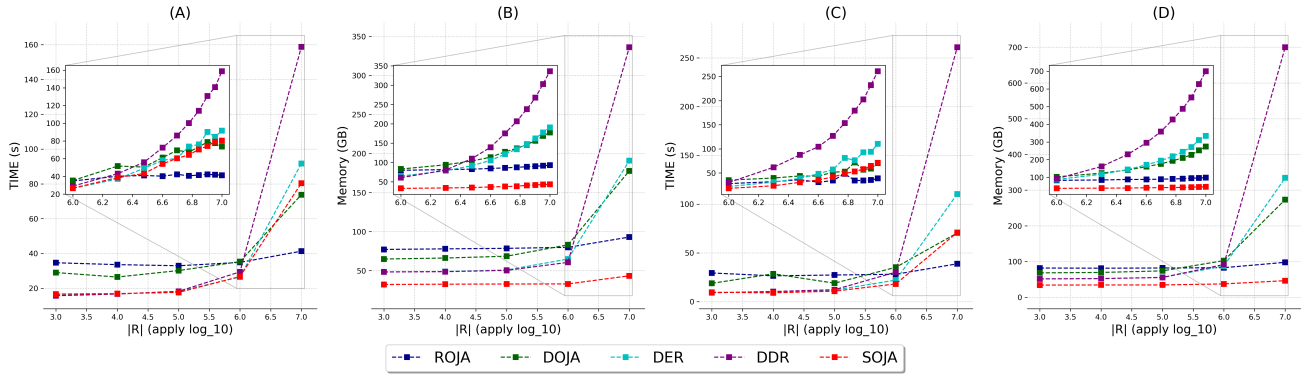


Figure 2: Execution time and memory usage against $|R|$, wide ranges $|R|$ from 10^3 to 10^7, narrow is from 10^6 to 10^7. (A) time vs N for 32; (B) memory usage vs N for 32; (C) time vs N for 64; (B) memory usage vs N for 64;

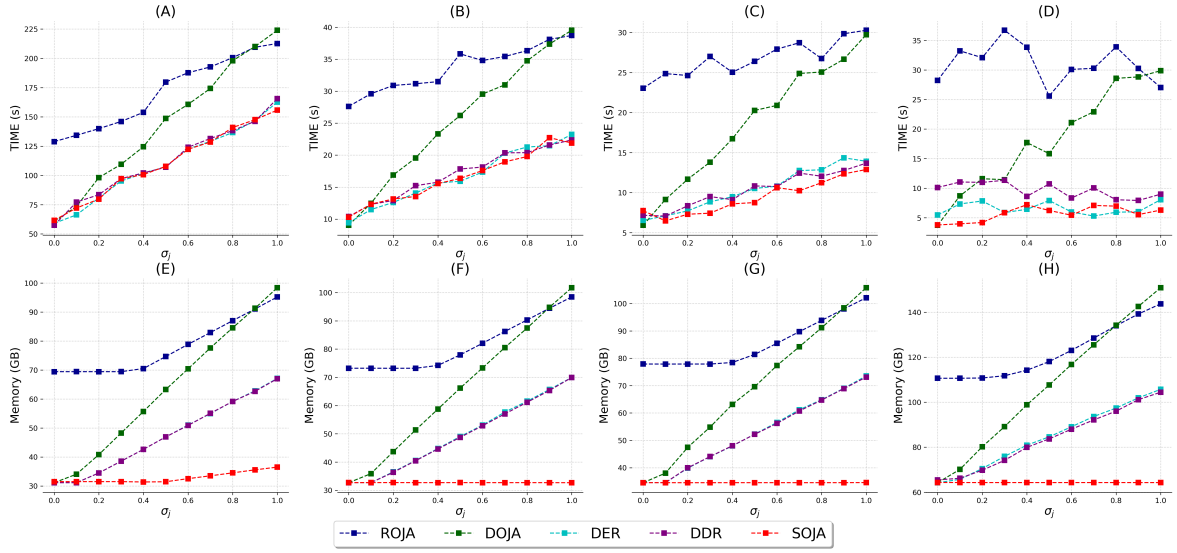


Figure 3: Execution time against selectivity ratio of left table over varying number of processes: (A - D): execution time for N = 4, 32, 64, 512; (E - H): memory usage for N = 4, 32, 64, 512;

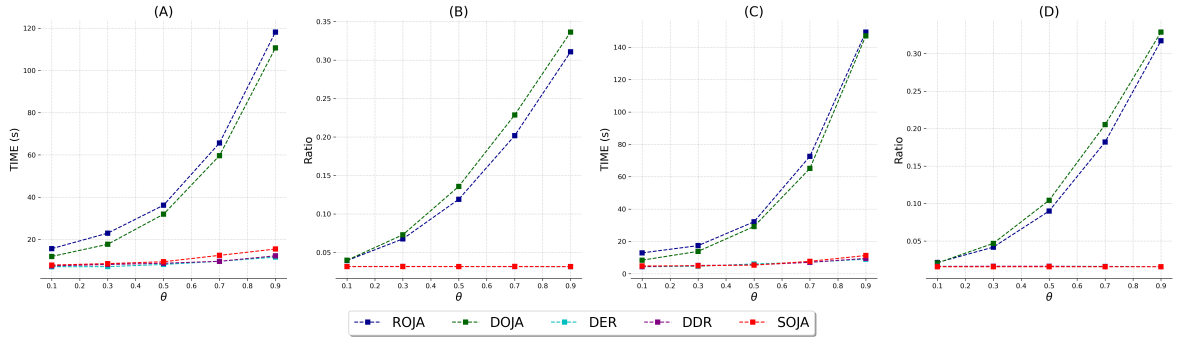


Figure 4: Execution time and memory usage ratio against skewness degree: (A) execution time for N = 32; (B) memory usage ratio N = 32; (C) execution time for N = 64; (D) memory usage ratio for N = 64;

Both execution time and memory usage in ROJA are not considerably affected by changes in $|R|$, because the cost of ROJA is dominated by the large table (although R increases, it is still at least 5 times smaller than $|S|$). The broadcast cost and subsequent local join between R and S_i in DOJA, DER and DDR increases with the growth of R, increasing both execution time and memory usage. Furthermore, dangling tuples candidates after the first local outer join will also increase as R increases. Therefore, DER and DDR have to redistribute and filter more data to identify non-matching tuples. The number of candidates is roughly

$|R| \times (1-\sigma_R) \times N$. Unlike DER which use ids, DDR directly works with tuples; therefore, its execution time and memory usage increases dramatically with an increase of $|R|$ and N. As for SOJA, the swap step becomes more expensive with the growth of $|R|$. Overall, we can see that when $|R|$ is 10 times less than $|S|$, DER, DDR and SOJA outperform ROJA in both time and memory.

4.3 Effect of Selectivity Ratio of Left Table

These experiments examine the algorithms' performance with different σ_R from 0.0 to 1.0 using a varying number of processes (4, 32, 64, 512). We set $|R|$ to 10^5 and $|S|$ as 5×10^7, and the σ_j to 10^{-5}.

The results in Figure 3 (A - D) show that the overall execution time has an upward trend with an increase of σ_R as a high σ_R leads to an increase of the result size and thereby increasing IO costs. The execution time of DOJA increases dramatically since DOJA has to redistribute and perform a second local join based on increasing inner join results. Overall, DER, DDR and SOJA share a similar performance which outperform the two conventional algorithms. The upward trend stabilizes with increasing number of processes due to reduction of local IO costs. Figure 3 (E - H) shows the change of memory usage with increase of σ_R. When σ_R is small, the redistribution cost in ROJA takes a considerable amount of memory, and it require even more memory to hold join results as σ_R increases. DOJA uses more memory compared to the other two broadcast algorithms because it redistributes the inner join results rather than the dangling tuples.

4.4 Effect of Data Skewness

To test the effect of skewness, we use the Zipf distribution model and the number of tuples for both tables after redistribution in the $i-th$ process ($|R_i^{redis}|$ and $|S_i^{redis}|$) is the following:

$$|R_i^{redis}| = \frac{|R|}{i^\theta * \sum_{j=1}^{N} \frac{1}{j^\theta}} \tag{8}$$

To generate data with skew, we sample multiple datasets with the join key using a Zipfian distribution with different degrees θ from 0.1 to 0.9. We set $|R|$ as 10^5, $|S|$ as 2×10^7, and σ_R as 0.5. In this experiment, we examine how execution time as well as memory usage changes on 32 and 64 processes with an increase of θ. In case of skewness, the execution time is determined by the process with the biggest workload. To make the memory usage results meaningful, we use a memory usage ratio, which is the maximum memory usage among all processes over the total memory usage, instead of total memory usage.

Experiments on both 32 and 64 processes (Figure 4) show that the degree of skewness has a significant impact on execution time and memory usage ratio of ROJA and DOJA due to unbalanced workload caused by the redistribution step. Although DER and DDR also feature a redistribution operation, their performance remains stable as θ changes. This is because, in this experiment, the number of dangling tuples/ids to redistribute is relatively small due to the small $|R|$. The memory ratio in Figure 4 (B & D) also reflects that both DER and DDR have a relatively balanced memory usage. Since this skewness is about data skew in the join key rather than the initial placement, both memory and execution performance of SOJA are not much affected by θ.

4.5 Discussion

ROJA, as a general-purpose outer join algorithm, does not have an outstanding performance when the left table's size is very small. Although ROJA is less affected by changes in the selectivity ratio in the left table, its performance with high data skew drops dramatically. Another conventional algorithm, DOJA, has no outstanding performance in most cases because it has to broadcast the entire left table and redistribute the entire inner results, which is particularly bad when the join selectivity ratio is large.

In small-large outer joins, DER shows an outstanding performance over most cases as it optimizes the redistribution by focusing on ids to identify dangling tuples. Using ids of dangling tuples makes DER less affected by the selectivity ratio and skewness. As a DER competitor, DDR adopts a simpler procedure, directly redistributing and filtering based on the dangling tuples itself. DER and DDR share the core methodology and their performance in both time and memory aspects is also similar in most cases. However, when the number of the tuples in the left table increases, the performance of DDR becomes the worst since the size of candidates of dangling tuple in bytes increases both computation and memory costs.

SOJA achieves a similar (sometimes better) performance than DER and DDR in terms of time costs in most cases and outperforms all competitors in terms of memory usage. Additionally, SOJA is less affected by data skewness and selectivity ratio since there is no redistribution operation in SOJA.

5 CONCLUSIONS

In this paper, we implemented four existing parallel outer join algorithms in MPI and proposed a new algorithm SOJA. SOJA does not simply optimize one specific part of existing algorithms; it provides an entirely novel approach, *swap*, to perform outer joins. The experiment based on HPC infrastructure shows that the performance of SOJA is outstanding, especially in memory usage. Further, we will investigate whether SOJA can be used in other joins, such as inner join, with the same benefits. Additionally, the feasibility of the swap approach in parallel spatial joins will be explored in future works.

REFERENCES

[1] Ablimit Aji and Fusheng Wang. 2012. High Performance Spatial Query Processing for Large Scale Scientific Data. In *SIGMOD/PODS '12 PhD Symposium.*

[2] Ablimit Aji, Fusheng Wang, and Joel H. Saltz. 2012. Towards Building a High Performance Spatial Query System for Large Scale Medical Imaging Data. In *SIGSPATIAL '12.*

[3] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed join algorithms on thousands of cores. *PVLDB* 10, 5 (2017).

[4] Spyros Blanas, Jignesh M Patel, Vuk Ercegovac, Jun Rao, Eugene J Shekita, and Yuanyuan Tian. 2010. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD '10.*

[5] Long Cheng, Ilias Tachmazidis, Spyros Kotoulas, and Grigoris Antoniou. 2017. Design and evaluation of small–large outer joins in cloud computing environments. *J. Parallel and Distrib. Comput.* 110 (2017), 2–15.

[6] Transaction Processing Performance Council. 2008. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html* 21 (2008), 592–603.

[7] TLC Trip Record Data. 2020. https://www1.nyc.gov/tlc.

[8] Victor Eijkhout. 2017. *Parallel Programming in MPI and OpenMP.* Lulu. com.

[9] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. 1999. *Using MPI: portable parallel programming with the message-passing interface.* Vol. 1. MIT press.

[10] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. 2014. *Using advanced MPI: Modern features of the message-passing interface.* MIT Press.

[11] Henry Markram. 2006. The Blue Brain Project. *Nature Reviews Neuroscience* 7, 2 (2006), 153–160.

[12] David Taniar, Clement HC Leung, Wenny Rahayu, and Sushant Goel. 2008. *High-performance Parallel Database Processing and Grid Databases.* Vol. 67.

[13] Fusheng Wang, Ablimit Aji, and Hoang Vo. 2014. High Performance Spatial Queries for Spatial Big Data: From Medical Imaging to GIS. *SIGSPATIAL Special* 6, 3 (2014).

[14] Yu Xu and Pekka Kostamaa. 2010. A new algorithm for small-large table outer joins in parallel DBMS. In *ICDE 2010.*

Industrial Paper

JENGA - A Framework to Study the Impact of Data Errors on the Predictions of Machine Learning Models

Sebastian Schelter*
University of Amsterdam
s.schelter@uva.nl

Tammo Rukat
Amazon Research
tammruka@amazon.com

Felix Biessmann†
Einstein Center Digital Future
fbiessmann@beuth-hochschule.de

ABSTRACT

Machine learning (ML) is increasingly used to automate decision making in various domains. Almost all common ML models are susceptible to data errors in the serving data (for which the model makes predictions). Such errors frequently occur in practice, caused for example by program bugs in data preprocessing code or non-anticipated schema changes in external data sources. These errors can have devastating effects on the prediction quality of ML models, and are, at the same time, hard to anticipate and capture.

In order to empower data scientists to study the impact as well as mitigation techniques for data errors in ML models, we propose Jenga, a light-weight, open source, experimentation library. Jenga allows its users to easily test their models for robustness against common data errors. Jenga contains an abstraction for prediction tasks based on a dataset and a model, an easily extendable set of synthethic data corruptions (e.g., for missing values, outliers, typos and noisy measurements) as well as evaluation functionality to experiment with different data corruptions.

Jenga supports researchers and practitioners in the difficult task of data validation for ML applications. As a showcase for this, we discuss two use cases of Jenga: studying the robustness of a model against incomplete data, as well as automatically stress testing integrity constraints for ML data expressed with tensorflow data validation.

1 INTRODUCTION

Many companies and organisations are moving to a data-driven approach, where machine learning (ML) is used to assist and automate decision making in various domains. Yet the application of ML in production settings often faces a number of pitfalls. Almost all common ML models are susceptible to data errors in the serving data (for which the model makes predictions). Such errors frequently occur in practice, caused for example by program bugs in data preprocessing code or non-anticipated schema changes in external data sources. These errors can have devastating effects on the prediction quality of ML models [12], and are, at the same time, hard to anticipate and capture. While many aspects of the impact of data changes on ML models are studied in the ML literature [1, 7, 9], it can be difficult to relate this research to the errors occurring in practical ML applications, as these approaches all require distributional assumptions about the change.

Data errors in production machine learning. Frequently, the errors in production deployments do not originate from changes in the data generating real-world processes, but from programming errors in the data pipelines constructing the serving data [4] or from errors during data integration from different sources [6, 21]. Such errors often only become apparent once models are deployed in complex production use cases [16].

We have come across several real world instances of such data errors. In one case, a linear model had been trained on demographic data (including a person's age), and the age value had been missing for some records for which the model should supply predictions. A software engineer (without knowledge of the model intricacies) then wrote preprocessing code to replace all missing age values with zeroes, the default value for initialising integers in many programming languages. This led to a unwanted misbehavior of the model, which effectively treated all these records as "toddlers". In another case, we learned about an ML model where the pipelines for training and serving data were running in different cloud environments. As a result, the code bases for data preparation on the training and serving side accidentally diverged, which introduced hard to detect data errors. Such errors can have devastating impact, as all guarantees about the reliability of the predictions of the ML model may be lost, which can lead to monetary losses (e.g., if buying decisions are made based on the predictions of a forecasting model) and bad user experiences (e.g., if users are presented with non-sensical recommendations in an online shop).

Evaluating the robustness of models against common data errors. These examples show the need for testing the robustness of ML models to data errors before they are deployed to production. Recent research focuses on detecting and handling such data errors, e.g., by proposing unit tests and integrity constraints for ML data [4, 17], ML-based missing value imputation [2] and validating the predictions of black box models [19]. In our experience, it is difficult to provide broadly valid empirical evaluations of these approaches, and to generate synthetically corrupted data that represents the scenarios that we encounter in the real world.

To address this need, we design the *Jenga* library, which we present in this paper. Jenga enables data scientists to study the robustness of their models against errors commonly observed in production scenarios. Based on the findings from experimenting with Jenga, users can take appropriate measures to protect their deployed models against impactful data errors, e.g., with custom integrity constraints implemented via tensorflow data validation [4].

In summary, this paper provides the following contributions.

- We introduce our open source framework Jenga to study the impact of data errors on ML models (Section 2).
- We describe how to implement custom prediction tasks and synthetic data corruptions in Jenga (Section 3).
- We discuss two use cases for Jenga: studying the robustness of a model against incomplete data, and automatically stress testing integrity constraints for ML data expressed with tensorflow data validation (Section 4).

*work done while at New York University
†work done while at Amazon Research and Beuth University, Berlin, Germany

Jenga is publicly available under an open source license at https://github.com/schelterlabs/jenga.

2 FRAMEWORK DESIGN

We introduce the design of Jenga. The goal of Jenga is to enable data scientists to evaluate the impact of data errors on their models, and to evaluate techniques that make these models more robust. We design Jenga around three core abstractions: (*i*) *tasks* contain a raw dataset, an ML model, and represent a prediction task; (*ii*) *data corruptions* take raw input data and randomly apply certain data errors to them (e.g., missing values); (*iii*) *evaluators* take a task and data corruptions, and execute the evaluation by repeatedly corrupting the test data of the task, and recording the predictive performance of the model on the corrupted test data.

We provide three sample tasks (Section 2.1), several data corruptions (Section 2.2) and two different evaluators (Section 2.3) as part of the framework.

2.1 Example Tasks

We provide three exemplary prediction tasks in Jenga. Note that users can define and implement their own custom tasks with low effort (see Section 3 for details). We choose simple binary classification tasks for product review classification (predicting whether the review of a video game was deemed helpful), income estimation (predicting whether a person earns more than $50,000 per year based on demographic data) and image recognition (distinguishing sneakers from ankle boots), which resemble real world use cases, and leverage publicly available datasets and widely used ML models. We focus on relatively small-scale problems, which do not require costly infrastructure (e.g., the models can be trained in a couple of minutes on a multicore CPU), in order to allow users to rapidly experiment and play with our framework. These tasks are meant as examples to enable users to test our data corruptions and evaluators, and serve as a template for our users to integrate Jenga with their own custom prediction tasks.

2.2 Data Corruptions

In the following, we describe the types of data corruptions available in Jenga. Each error type requires the specification of a column c to be affected by the error and a fraction of rows $r \in [0, 1]$ that should be affected.

Corruption sampling.: Whether or not a value is affected by a corruption is often the result of errors in complex preprocessing pipelines. In order to account for realistic corruption patterns we model the fraction of rows affected by a corruption as follows. A value x_c in column c is corrupted (*i*) *independent of other values* (corrupted values are sampled *completely at random*), (*ii*) dependent on values in columns other than c (corrupted values are sampled *at random*), or (*iii*) dependent on values in column c (corrupted values are sampled *not at random*). This modelling is inspired by literature on missing value imputation, where three types of missingness are commonly distinguished [10]. As these three sampling procedures a can capture the complex error patterns often observed in practice we chose to make it applicable not only to missing value corruptions, but to all other error types as well.

Missing values. Missing values are amongst the most common data errors in practice. Missing values can have devastating effects on training and prediction, depending on how a data pipeline deals with missing values before feeding the data to a downstream ML model. An important factor for the impact of missing values are the missingness patterns, described in the previous paragraph, *missing completely at random (MCAR)*, *missing at random (MAR)* and *missing not at random (MNAR)*.

We additionally support the injection of *missing values based on "prediction difficulty"*, where we consider the fact that there ML model downstream that is affected by missing data. This error type considers the entropy of the ML model predictions for the data rows and discards values based on their difficulty for the model, akin to uncertainty sampling in active learning.

Swapped values. We replace a specified ratio of values in one column with values in another column. This corruption mimics users mixing up entries in input forms [6] or programming errors in data preparation code, where a programmer accidentally swaps target columns to write to.

Scaling. We randomly scale a subset of the values by 10, 100 or 1000. This perturbation mimics cases where the scale of an attribute is accidentally changed in preprocessing code (e.g., because a developer accidentally changes the code to record durations in milliseconds instead of seconds).

Noise. We corrupt a fraction of a column's values by adding gaussian noise centered at the data point with a standard deviation randomly selected from the interval of 2 to 5. This corruption is intended to mimic measurement errors.

Encoding errors. This corruption replaces certain characters in string attributes (e.g., a with á), and is meant to simulate encoding errors, e.g., for data retrieved from web pages which indicate a false encoding.

Image corruptions. Dealing with corrupted training images is a well studied problem in computer vision [3] for which a lot of tooling exists already. We therefore integrate existing image corruptions from the *augmentor*[1] library into jenga.

2.3 Evaluators

Finally, Jenga provides so-called *evaluators*, which measure the impact of data corruptions on the model's predictive performance. Jenga currently features two evaluators: The `CorruptionImpactEvaluator` takes a provided task, a trained model and a manually specified list of corruptions. It applies each data corruption to the held-out test set of the task and computes the predictive performance of the model in light of the data corruption. We show how to use this evaluator with a few lines of code in Section 3.1, and discuss a detailed example of measuring the impact of missing values on a task in Section 4.1.

The second evaluator allows users to additionally integrate a data validation schema into the evaluation. In many cases, it is not possible to make an ML model completely robust against data errors [19]. A common approach to prevent feeding corrupted data to deployed ML models is to run data validation checks on the serving data on which the model is applied. Popular libraries for this task are *tensorflow data validation (TFDV)*[2] [4] or Deequ[3] [18]. They allow users to define a schema and constraints for the serving data (e.g., that a given attribute must not contain missing values), and efficiently execute this check before passing the data to an ML model. Jenga contains a custom `SchemaStresstestEvaluator` for feature data validation with a TFDV schema. This evaluator works analogous to the previous

[1] https://github.com/mdbloice/Augmentor
[2] https://www.tensorflow.org/tfx/guide/tfdv
[3] https://github.com/awslabs/deequ

one, but additionally records whether the check of a provided data validation schema would have correctly detected the negative impact of the data corruption. We provide an extensive example for this evaluator in Section 4.2.

3 USAGE AND CUSTOMISATION

We implement Jenga based on several popular open source ML frameworks in python. We leverage *pandas* for data wrangling, and *numpy* for numerical computations. We implement feature extraction and preprocessing via *scikit-learn*'s pipeline abstraction, and also use classical ML models from this library. We rely on *keras* and *tensorflow* for defining and training neural networks.

In the following we first give an example of how to use Jenga to evaluate the impact of data corruptions (Section 3.1), and subsequently discuss how to implement custom tasks and data corruptions in Jenga's API in Section 3.2.

3.1 Evaluating the Impact of Data Errors

The core use case of jenga is to evaluate the impact of certain data corruptions on a prediction task. This can be implemented with a few lines of code: We have to instantiate the task and data corruptions that we want to evaluate, and can execute the evaluation with Jenga's `CorruptionImpactEvaluator`. This allows us to measure the impact of a predefined list of data corruptions on the predictive performance of a model.

```
# Create the prediction task
task = IncomeEstimationTask()
# Train a baseline model
model = task.fit_model(task.train_data,
 task.train_labels)
# Specify the data corruption to test
corruption = MissingValues(column='age',
 missingness='mcar', fraction=0.05)
# Create the evaluator
evaluator = CorruptionImpactEvaluator(task)
# Run the evaluation with 10 repetitions
result = evaluator.evaluate(model,
 num_repetitions=10, corruption)
# Impact on predictive performance
print(f""" Score on
  clean data:     {result.baseline_score}
  corrupted data: {result.corrupted_scores} """)
```

Here, we setup a task, train the corresponding model and define the corruption that we are interested in. We provide these to the evaluator together with the specification of the number of repetitions to execute for each corruption. The evaluator repeatedly corrupts the (copied) test data of the task, computes the prediction quality of the model on the corrupted data and finally provides a result object with the corresponding scores for each corruption to investigate. Note that we could also have specified more than one data corruption to evaluate.

3.2 Custom Tasks and Data Corruptions

We design Jenga with the goal to make it easy for data scientists to wrap their existing code as a prediction task, which allows them to reuse our data corruptions and evaluators. In addition, we also make it easy to design custom data corruptions. Therefore, we next describe how to implement the two basic building blocks of Jenga, a `Task` and a `DataCorruption`.

Implementing a custom task. Jenga allows data scientists to implement custom tasks with low effort. We provide an abstract base class `ClassificationTask` with two methods that users must implement. In the constructor, users have to load the input data for the task. Next, they have to implement the `fit_model` method, which trains the accompanying prediction model for the task from training data provided in a pandas dataframe. The model produced by this must support scikit-learns predictor API. Finally, the `score_on_test_data` must be implemented, which computes the desired metric for the task (e.g., ROC AUC) from the predicted label probabilities of the model.

Implementing a custom data corruption. At the core of Jenga are data corruptions, whose impact on the predictive performance of a model we want to investigate. Data corruptions transform a dataframe into another dataframe with potentially corrupted values. We provide an abstract base class, `DataCorruption`, that users can extend by providing only a single method, called `transform`. In the following listing, we implement a data corruption that mimics a case where duration that needs be expressed in seconds is accidentally recorded in milliseconds (e.g., scaled by a factor of 1000) in a fraction for the rows.

```
class MillisInsteadOfSeconds(DataCorruption):
 ...
 def transform(self, data):
  # Operate on a copy of the data
  corrupted_data = data.copy(deep=True)
  # Pick a random fraction of the rows
  rows = np.random.uniform(len(data)) < self.fraction
  # Multiply the column values of the chosen rows
  corrupted_data.loc[rows, self.column] *= 1000
  return corrupted_data
```

We first conduct a deep copy of the input data, which we will corrupt later on. Then, we randomly pick the indexes of the rows that we want to corrupt, and finally mulitply their values by a 1000 to mimic milliseconds. Note that tasks and data corruptions implemented with our API can be readily used in the existing evaluators from Jenga, as outlined in Section 3.1.

4 EXAMPLE USE CASES

We discuss two examplary use cases of our framework that resemble real world problems which we encountered in production ML applications. Note that we provide implementations (in the form of Jupyter notebooks) for all these use cases in our github repository at https://github.com/schelterlabs/jenga.

4.1 Measuring the Robustness of a Model against Incomplete Data

Overview. In our first experiment, we showcase how to study the impact of missing values on the predictions of a model. This targets a common usage scenario, where data scientists, who have a trained model in production (or ready for production), want to study its robustness towards incomplete data with Jenga. They want to reach a conclusion on how well the model itself (in combination with different missing value imputation methods) can mitigate the impact of missing values in the serving data. Incomplete data is a very common issue in real world deployments, where data is often missing as a result of programming errors, data integration issues or unanticipated schema changes in an external data source.

Setup. We experiment with a logistic regression model for our income estimation task from Section 2.1. The goal of this task is to predict from demographic data whether an individual has a high income. We train a model on clean training data, and evaluate its predictive performance (in terms of ROC AUC) on test data with synthetically injected missing values. We focus on four categorical attributes in the data: `education`, `marital_status`,

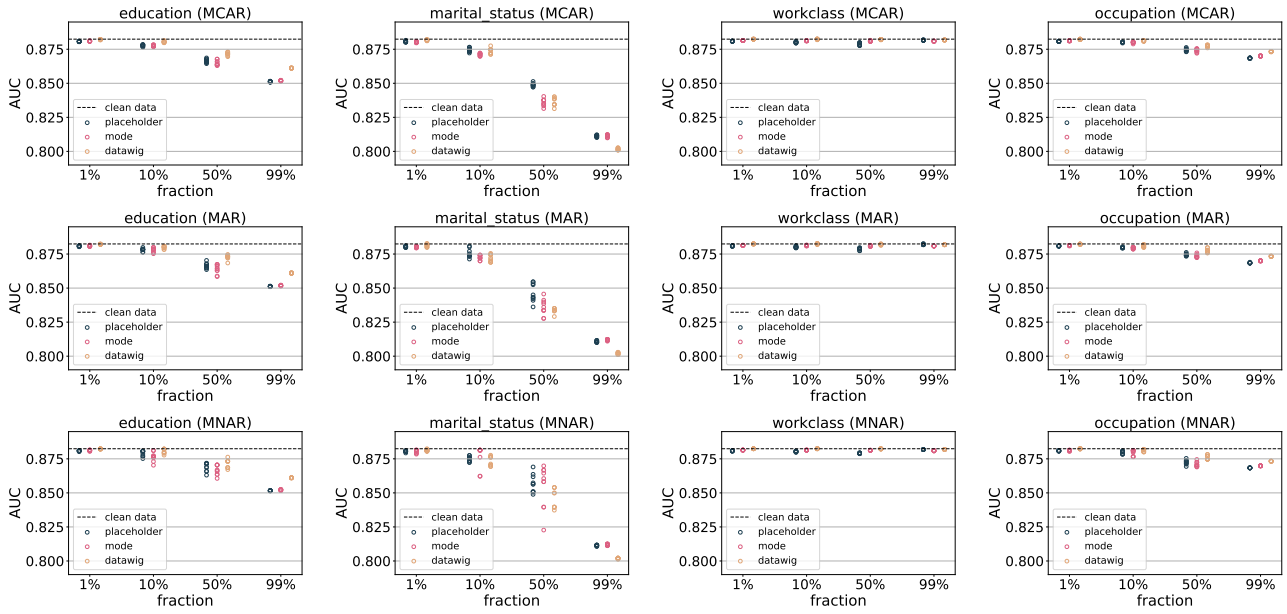


Figure 1: Evaluation of the robustness of a model for the income estimation task against incomplete data. We plot the AUC score achieved with different missing value imputation strategies (*placeholder*, *mode* and *datawig*) against the fraction of injected missing values. The impact differs by attribute and there is no clear dominating imputation strategy, indicating that it is difficult to make the model fully robust against this type of data error.

`workclass`, and `occupation`, and inject missing values into 1%, 10%, 50% and 99% of randomly chosen values of a given attribute.

We repeat this process for all three kinds of missing values ("missing completely at random" (MCAR), "missing at random" (MAR), "missing not at random" (MNAR)) as discussed in Section 2.2. We repeat each individual configuration ten times, and report the performance on corrupted test data (in comparison to the performance on clean data), where we differentiate between three different ways to make the model handle the missing values:

- First, we replace missing values with a constant *placeholder* symbol.
- Secondly, we replace missing values with the *mode* (the most frequent value in the column) via scikit-learn's `SimpleImputer`
- Thirdly, we train a dedicated ML model to impute missing values based on the structure present in the complete records [2]. We leverage the *datawig* library[4], which automatically featurises tabular data and trains a neural network to predict the missing values.

Results. The experimental results are shown in Figure 1. We find that the impact of the missing values is highly dependent on the attribute we target. There is nearly no impact for `workclass`, a very minor impact for `occupation` for less than 50% missing values, a much stronger impact for `education`, and we encounter the highest impact for missing values in `marital_status`. We additionally see that the impact is in some cases different for different types of missing values, e.g., values "missing not at random" in the `marital_status` attribute seem to be easier to handle than the other types of missingness.

In summary, we find no clear dominating strategy for handling the missing values in this particular task. Having the model deal with the missing values via a placeholder symbol is simple and works well in many cases. However, there are some setups where leveraging a dedicated missing value imputation strategy helps, e.g., datawig for a high fraction of missing values in `education` or for `occupation`. We conclude that the model itself cannot handle missing values reliably in all cases, even in combination with imputation. Thus, the data scientists need to put checks in place to safeguard the serving data on which the model is applied.

4.2 Stresstesting Integrity Constraints for ML data

Overview. Our next experiment shows how to put safeguards in place for an ML model. This experiment applies a schema and constraints for ML data, and executes a stresstest for them, as discussed in Section 2.3. We leverage the product review classification task discussed in Section 2.1, where the goal is to predict whether uses found the review of a videogame helpful or not.

We train a model for this task, and additionally create a schema with integrity constraints for the test data in TFDV. Next, we run Jenga's `SchemaStresstest` which generates random data corruptions for the test data, and determines whether our schema catches these errors, and what the impact of these on the prediction quality of the model (in terms of ROC AUC) would have been.

In real world use cases, it is difficult for data scientists to come up with an appropriate schema and constraints for their data, and we develop our stresstest to uncover errors which are not caught by the current schema. As a consequence, data scientists can iteratively improve their integrity constraints until they pass the stresstest.

[4]https://github.com/awslabs/datawig

```
from jenga.tasks.reviews import VideogameReviewsTask
from jenga.evaluation.schema import SchemaStresstest
import tensorflow_data_validation as tfdv
# Setup task
task = VideogameReviewsTask()
# Create a schema to test
train_data_stats =
 tfdv.generate_statistics_from_df(task.train_data)
# Auto-infer schema from training data
schema = tfdv.infer_schema(statistics=train_data_stats)
# Manually adjust schema
review_date_feature =
 tfdv.get_feature(schema, 'review_date')
review_date_feature.distribution_constraints
 .min_domain_mass = 0.0
# Define model to include in stress test
model = task.fit(task.train_data, task.train_labels)
# Run stress test with 250 randomly generated
# data corruptions
stress_test = SchemaStresstest()
results = stress_test.run(task, model, schema,
 num_corruptions=250, performance_threshold=.03)
```

Setup. The code above shows the setup of the experiment. We generate a schema for the feature data of the task, with a semi-automatic approach, where we first have TFDV automatically infer a schema for the data (via `tfdv.infer_schema`).

The schema correctly identifies the data types and categorical domains of most of the attributes of the data. It is too strict however, as it does not account for the fact that all the values in the `review_date` column will change for future data. We manually adjust the schema for this attribute by setting the minimum domain mass that must be shared between the values found at schema inference time and the future values to zero, allowing new values to appear in the column. The following listing shows an excerpt of the schema and constraints for the data, containing type information, completeness requirements and domain values for the data attributes.

```
...
feature {
 name: "star_rating"
 type: INT
 presence { min_fraction: 1.0 } }

feature {
 name: "verified_purchase"
 type: BYTES
 domain: "verified_purchase"
 presence { min_fraction: 1.0 } }

feature {
 name: "review_date"
 type: BYTES
 domain: "review_date"
 presence { min_fraction: 1.0 }
 distribution_constraints { min_domain_mass: 0.0 } }
...
string_domain {
 name: "verified_purchase"
 value: "N"
 value: "Y"
}
...
```

We evaluate the schema with a stress test which applies 250 randomly generated data corruptions to the serving data of the model and measures their impact on the prediction quality.

Results. The model achieves an AUC of 0.78828 on clean data, and we consider all predictions on corrupted data with more than 3% decrease in prediction performance as failures. Jenga categorizes the results as following:

- True positives, where TFDV reports a schema violation and the prediction quality on the corrupt test data drops below the threshold.
- True negatives, where TFDV reports no schema violation and the prediction quality on the corrupt test data is within the threshold.
- False positives, where TFDV reports a schema violation, but the prediction quality on the corrupt test data does not drop below the threshold. Note that it might still make sense to capture and investigate these data errors, as they can be indicators of problems in preprocessing code or external data sources.
- False negatives, where TFDV reports does not report a schema violation, but the prediction quality on the corrupt test data does drop below the threshold. These are the most important findings from a stress test as they indicate data errors to which the model would be vulnerable in production. It is imperative to adjust the schema to catch these errors.

In the following, we list several findings from our stress test example in Table 1 and discuss them.

True positives. Out of the 250 corruptions, we find 88 true positives. For example, we find that the model crashes for missing values in the numeric `star_rating` column, and that the prediction quality drops more than 3% for gaussian noise in this column and for a large number of swapped values between the `verified_purchase` and `title` column. Note that all of these errors are correctly detected by TFDV.

error type	*column(s)*	*frac*	*comment*
True positives			
missing values	`star_rating`	.25	crash
swapped values	`review_body, vine`	.75	unseen values
swapped values	`verified_purchase, title`	.45	unseen values
missing values	`vine`	.53	incompleteness
gaussian noise	`star_rating`	.25	type (int to float)
True negatives			
encoding	`vine`	.72	no changes
encoding	`review_id`	.83	column not used
swapped values	`review_id, product_parent`	.17	columns not used
missing values	`product_id`	.27	column not used
False positives			
missing values	`vine`	.93	unseen values
gaussian noise	`star_rating`	.16	type (int to float)
swapped values	`product_id, marketplace`	.35	unseen values
encoding	`marketplace`	.72	unseen values
False negatives			
scaling	`star_rating`	.92	range check missing
encoding	`title_and_review`	.76	no encoding checks
missing values	`title_and_review`	.80	no length checks
swapped values	`title_and_review, review_headline`	.75	no length checks

Table 1: Results found by the schema stress test for detecting impactful data errors on the product review task.

True negatives. We additionally find 75 true negatives, which mostly include cases where a textual column is being corrupted which is ignored by TFDV, but also not used by the model, whose prediction quality is therefore not affected by the corruption.

False positives. We encounter 39 false positives. We for example see that even a high number of missing values in the `vine` column

do not strongly affect the prediction quality, as well as a low number of noisy values in the `star_rating` column or encoding errors in the `marketplace` column, which is not used by the model.

False negatives. The most important results from the stress test are false negatives, e.g., data corruptions that are not detected by our TFDV schema, but strongly affect the prediction quality of the model. In a real world use case, we need to extend our schema to catch all these errors. We see that scaling values in the `star_rating` column strongly affects the prediction quality. This is an indicator that we should add a range check for this column to our schema. Furthermore, all kinds of errors in the `title_and_review` column negatively affect the prediction quality. This is a textual column for which TFDV does not generate constraints automatically. Checks for both the length and encoding of the values in that column are required to capture the outlined errors.

We argue that it should become a best practice to execute such stresstests for data errors before putting ML models into production, and we think that such testing capabilities should be integrated into common ML deployment pipelines.

5 RELATED WORK

Addressing the challenges in productionizing ML models is a field with growing interest in recent years [2, 8, 12, 16, 20]. Several solutions were proposed for validating ML models and their predictions. Most of these originate from a statistical ML or a data management perspective. Approaches from the ML community are based on distributional assumptions about the data shift, such as label shift [13], and covariate shift [1]. These assumptions often seem inapt to describe practically relevant data changes for engineers, such as the errors described above. Moreover, the proposed methods often limit themselves to adapting a particular model or learning paradigm.

There exist several approaches from the data management community to validate the input data of ML pipelines. For example, Google's TFX platform [4] offers validation for input data via a feature schema, and Deequ [17] enables unit tests for data, but both of them do not quantify the potential impact of errors on the model predictions. On a related note, there is a growing body of work on model monitoring [19], model diagnosis [5] and model unit testing for neural networks [11].

6 LEARNINGS & CONCLUSION

During our work on real world ML deployments, we have repeatedly come across scenarios where data errors heavily impacted deployed models and applications.

Missing values in data can in some cases, propagate through various connected pipelines until a customer facing model crashes, and it is very tedious to trace these errors back to the original data source which introduced the missing values. In internationalised applications, which operate on text in non-western languages, it is common to encounter encoding issues which are often introduced by a wrongly configured intermediate data store, and are again very hard to pinpoint and fix. Another common source of errors is calendar-related data, where dates and durations are often incorrect, e.g., due to movable holidays. Furthermore, often ML models are trained by specialised teams, and then handed over to business teams. In such cases, we often experienced that the data provided to the ML experts had not been sampled in a representative way by the business team, and as a consequence, the resulting model will not perform well later on due to the dataset shift introduced by the non-representative sampling.

These experiences motivate our presented library Jenga, which enables data scientists to evaluate the performance of ML models under data errors. Jenga builds on existing ML libraries, and allows practitioners and researchers to quickly build ML testing suites for their models with a broad range of data errors that we observed over several years of maintaining ML applications. We think that is is necessary to establish a set of best practices for testing ML models, analogous to established best practices like unit testing and integration test in software engineering. The goal of Jenga is to collect a huge library of data corruptions that occur in the real world, and uses these to automate the testing of ML models, ideally with an integration into upcoming systems for continuous integration for ML [14].

In the future, we aim to extend Jenga to extend more diverse tasks (e.g., regression problems or ranking problems). We will continue to work on Jenga as part of our recently proposed vision for automated ML model monitoring with respect to data quality [15]. We hope that Jenga can contribute to future research on data governance for end-to-end management platforms for ML.

REFERENCES

[1] Steffen Bickel, Michael Brückner, and Tobias Scheffer. 2009. Discriminative learning under covariate shift. *JMLR* 10, 2137–2155.

[2] Felix Biessmann, David Salinas, Sebastian Schelter, Philipp Schmidt, and Dustin Lange. 2018. Deep Learning for Missing Value Imputation in Tables with Non-Numerical Data. In *CIKM*. 2017–2025.

[3] Marcus D Bloice, Peter M Roth, and Andreas Holzinger. 2019. Biomedical image augmentation using Augmentor. *Bioinformatics* 35, 21 (2019), 4522–4524.

[4] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *SysML*.

[5] Yeounoh Chung, Tim Kraska, Steven Euijong Whang, and Neoklis Polyzotis. 2018. Slice finder: Automated data slicing for model interpretability. *SysML*.

[6] Joseph M Hellerstein. 2008. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)* (2008).

[7] Jiayuan Huang, Arthur Gretton, Karsten M Borgwardt, Bernhard Schölkopf, and Alex J Smola. 2007. Correcting sample selection bias by unlabeled data. *NeurIPS*, 601–608.

[8] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2020. CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks. In *ICDE*.

[9] Zachary C Lipton, Yu-Xiang Wang, and Alex Smola. 2018. Detecting and Correcting for Label Shift with Black Box Predictors. *ICML*.

[10] R. J. A. Little and D. B. Rubin. 2002. *Statistical analysis with missing data. 2nd ed.* Wiley-Interscience, Hoboken, NJ,.

[11] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. *SOSP*, 1–18.

[12] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data Lifecycle Challenges in Production Machine Learning: A Survey. *SIGMOD Record* 47, 2, 17.

[13] Stephan Rabanser, Stephan Günnemann, and Zachary Lipton. 2019. Failing loudly: an empirical study of methods for detecting dataset shift. In *NeurIPS*. 1394–1406.

[14] Cedric Renggli, Frances Ann Hubis, Bojan Karlaš, Kevin Schawinski, Wentao Wu, and Ce Zhang. 2019. Ease. ml/ci and Ease. ml/meter in action: towards data management for statistical generalization. *VLDB* 12, 12 (2019), 1962–1965.

[15] Tammo Rukat, Dustin Lange, Sebastian Schelter, and Felix Biessmann. 2019. Towards Automated ML Model Monitoring: Measure, Improve and Quantify Data Quality. *ML Ops workshop at MLSys* (2019).

[16] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. 2018. On Challenges in Machine Learning Model Management. *IEEE Data Engineering Bulletin* 41.

[17] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification. *PVLDB* 11, 12, 1781–1794.

[18] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1781–1794.

[19] Sebastian Schelter, Tammo Rukat, and Felix Biessmann. 2020. Learning to Validate the Predictions of Black Box Classifiers on Unseen Data. *SIGMOD*.

[20] D Sculley et al. 2015. Hidden technical debt in machine learning systems. *NeurIPS*, 2503–2511.

[21] Michael Stonebraker and Ihab F Ilyas. 2018. Data Integration: The Current Status and the Way Forward. *IEEE Data Eng. Bull.* 41, 2 (2018), 3–9.

Industrial Paper

Decongestant: A Breath of Fresh Air for MongoDB Through Freshness-aware Reads

Chenhao Huang
The University of Sydney
chenhao.huang@sydney.edu.au

Michael Cahill
MongoDB Inc
michael.cahill@mongodb.com

Alan Fekete
The University of Sydney
alan.fekete@sydney.edu.au

Uwe Röhm
The University of Sydney
uwe.roehm@sydney.edu.au

ABSTRACT

Many distributed databases deploy primary-copy asynchronous replication, and offer programmer control so reads can be directed to either the primary node (which is always up-to-date, but may be heavily loaded by all the writes) or the secondaries (which may be less loaded, but perhaps have stale data). One example is MongoDB, a popular document store that is both available as a cloud-hosted service and can be deployed on-premises. The state-of-practice is to express where the reads are routed directly in the code, at application development time, based on the programmers' imperfect expectations of what workload will be applied to the system and what hardware will be running the code. In this approach, the programmers' choice may perform badly under some workload patterns which could arise during run-time. Furthermore, it might not be able to utilize the given resources to their full potential – meaning database customers pay more money than needed.

In this paper, we present Decongestant: a system which will automatically and dynamically, as the application is running, choose where to direct reads, sending enough reads to secondaries when this will reduce load on a congested primary and boost the performance of the database as a whole, but without exceeding the maximum data staleness that the clients are willing to accept. A central insight is to use measured latency of read operations on primary and secondaries to determine whether the primary is congested.

In an experimental evaluation, we demonstrate our system adapts well to dynamically changing workloads, obtaining performance benefits when they can arise from use of the secondaries, while ensuring that returned values are fresh enough given client requirements. Our approach is decentralised and can be used by both cloud-consumers and on-premises users: it uses only client observations and the limited diagnostic data provided by the database to its clients.

1 INTRODUCTION

Distributed databases have become a popular offering due to their scalability and elasticity. Distributed databases typically deploy a combination of data sharding and replication over multiple nodes to provide both scalability and availability. Thus, there are typically multiple copies of data, and some distributed database offerings expose those to programmers via a performance tuning parameter where one can decide to which node (primary or secondary) the read requests should get routed. The challenge is to use this tuning knob wisely to balance the load, for good performance, while minimizing its pitfalls, namely access to potentially stale data. One typical example in this space is MongoDB, a popular NoSQL distributed database management system. MongoDB Atlas[1] is the corresponding cloud-hosted MongoDB service.

MongoDB internally is a classic log-based, primary-copy replication system. It is usually run as a replica set, where each node keeps a logical copy of the database [35]. Each replica set is a log-replicated state machine [34]. In cloud settings, all nodes are usually placed within one geographical region, but spread in different availability zones. There is one primary copy which processes all write operations. Each secondary copy pulls the updated log from the primary and then replays it to keep up with the primary. Thus, data on the secondary copies might be stale compared to that on the primary copy in a workload with frequent write operations. During a fail-over, one secondary copy is elected as the new primary.

Clients can direct read operations to either the primary copy or to a secondary copy (this is called *Read Preference* [24] and is indicated in the API as an optional parameter of a read request call). When reading from the primary copy on a healthy cluster with default settings, fresh data is returned. But if the primary copy is saturated, the read latency can be huge. In that case, a user can gain larger throughput and lower read latency by reading from secondary copies; however, data returned might be stale.

The state-of-the-art practice is to "hard-code" the Read Preference, making a choice explicitly when writing the application program. This is not ideal for several reasons. Firstly, developers may have insufficient information about workload and hardware capabilities for them to make a sensible choice when the code is written. Also, the "sensible" choice may differ over time as workload changes, but hard-coding is not able to adapt dynamically. Furthermore, the standard Read Preference options are limited to either primary or secondary, so that the nodes in a MongoDB cluster will never do the heavy-lifting together.

This paper shows how to capture knowledge *at run-time* of the current condition, and change the Read Preference dynamically, so that the overall performance of MongoDB can be improved. We adjust the proportion of secondary reads sent, in order to distribute work among the nodes of a MongoDB cluster. Our goal is to gain extra performance by reducing problems of congested nodes, so that the database can serve more clients without upgrading its hardware. This should help lower the cost of MongoDB for users. But we need to ensure each client sees data values that are "fresh enough" for the client's requirements, so we must avoid directing reads to a secondary when that node is too stale.

Achieving this is not easy. We must make the decision on where to direct a read on client-side information. Our approach

[1] https://www.mongodb.com/cloud/atlas

aims at both MongoDB cloud-consumers and on-premises users – there is no need to alter server code, or monitor the database's internal state. We only take measurements from the outside, as operations are submitted, or call the service API for the limited statistics it provides on the current status of servers. Our key insight to overcome this challenge is to measure the latency of read operations that were recently submitted by the client, for those sent to the primary and also for those sent to the secondary. In this paper we look at reads individually; if the client also needs session properties such as read-your-own-writes, or transaction-respecting snapshots, those can be obtained through capabilities of MongoDB, e.g. causal consistency[37].

Our key contributions are

- A mechanism to determine at run-time as either a cloud-consumer, or an on-premises user, whether the primary, or the secondaries, are currently congested with reads. This uses client-side measurements of the latency of recent read operations with each Read Preference, estimates the server-side component of that by subtracting network round-trip latency, and compares these estimates from the primary versus from the secondaries. This ratio is used in a simple feedback control. A high ratio indicates too much load on the primary, so more reads should go to the secondaries as long as those are not too stale. On a low ratio, indicating too much load on the secondaries, fewer reads should be sent to the secondaries in future.
- A system design and prototype implementation, called Decongestant, which uses this mechanism. It chooses at run-time to send a client's reads to primary or secondary as appropriate, for good performance while respecting a client-assigned limit on acceptable staleness of values returned.
- Experimental evaluation of Decongestant, to show that it maintains good performance while workloads vary (and it can do better than either hard-coded approach), and that it respects a client-assigned limit on staleness.

Our earlier poster paper [21] presented initial ideas in this direction. Here we adapt some of the text from there, describing motivation, background, and related work. Novel features of Decongestant, compared to [21], are: avoiding use of a secondary when it would give values that are too stale for client requirements (and never send reads to the secondaries when the clients can not tolerate any stale reads); capability to deliberately, in a controlled way, mix use of primary and secondaries in the same period, in order to outperform either hard-coded approach; improved latency estimate that separates network effects from those of server congestion; evaluations that show run-time adaptation to workloads that change in various aspects; a new S workload we use to demonstrate the validity of staleness reports from the servers (which Decongestant uses) as estimate of client-observed staleness.

The remainder of the paper is structured as follows. Section 2 introduces relevant concepts in MongoDB. Section 3 presents the design of Decongestant. We provide an evaluation of Decongestant in Section 4. Section 5 highlights, and contrasts with, related work. We conclude in Section 6.

2 BACKGROUND

Some text in this section already appeared in [21].

2.1 A brief overview of MongoDB and MongoDB-as-a-Service

As we have already discussed in section 1, MongoDB internally uses asynchronous primary-copy replication for fault-tolerance. This means it usually runs as a replica set. Fail-overs are rare [35]. This justifies our approach that exploits some of the capacity of the secondary nodes.

MongoDB also provides a sharding mechanism to enable horizontal scaling [22]. The entire database can be sharded into a number of shards, where each shard is a distinct subset of the whole database. Each shard can be deployed as a replica set, geographically far away from one another. MongoDB sharding is not used in this paper but the techniques we describe can be applied to sharded clusters, which support the same Read Preference API as standalone replica sets.

MongoDB Atlas is a "containerized" version of MongoDB, provided as a service. MongoDB Atlas has the same core as the open source Community Server, but with some additional, closed source functionality (e.g., security features). Customers are able to use MongoDB Atlas in a pay-as-you-go model. MongoDB Atlas is hosted on diverse cloud service platforms: Amazon Web Service (AWS), Microsoft Azure, or Google Cloud Platform (GCP). Customers can select the service locations, memory and storage size, number of vCPUs, etc. The database can be deployed with a few clicks. It is also trivial to scale-up if more computational power or storage are needed. MongoDB Atlas has various APIs for different programming languages and applications.

The main source of operational metrics offered by both MongoDB and MongoDB Atlas is via calling MongoDB `serverStatus` command. The MongoDB metrics can be queried frequently, but these deal only with internal properties, and do not report on hardware/OS/network aspects. MongoDB Atlas provides extra sources of operational metrics through MongoDB Atlas web API. This supplies some hardware metrics. However, those metrics only update once per minute, and can only be queried at a limited rate (100 requests per minute per project).

2.2 Read Preference

The Read Preference setting on read operations determines where they will be sent by MongoDB clients [24]. Read Preference options include `primary` (default), `primaryPreferred`, `secondary`, `secondaryPreferred`, and `nearest`. If the Read Preference `primary`, or `secondary`, is selected, then the reading request is directed to the primary copy, or to one of the secondary copies, respectively. Note that in MongoDB the users can not specify which of the secondary copies a read request will be sent to. `PrimaryPreferred` and `secondaryPreferred` provide users the option that in most cases the reads are sent to the primary or the secondary copies. However, in the situations where the preferred copy is not available, the reads are routed to the other option. When Read Preference `nearest` is chosen, the reading requests are directed to the nearest copy to the client based on client-measured network latency. The MongoDB client libraries periodically check which node is nearest. In our research, we use options `primary` and `secondary` to balance the workload among all MongoDB nodes.

When requesting a read on secondary copies, the clients can include a `maxStalenessSeconds` value to specify the maximum data staleness the client is happy to accept [25]. However, the `maxStalenessSeconds` value must be set to 90 seconds

or larger. As we will show later, Decongestant is able to bound the data staleness to substantially lower levels (eg 10 seconds).

The client driver randomly chooses a secondary copy to route a secondary read, as long as the latency between the secondary nodes do not differ by more than 15 milliseconds [28]. In our experiments, all secondary reads are directed to one secondary copy randomly.

There is another tuning knob in MongoDB called Read Concern which determines the durability, consistency, and isolation properties of the data read from MongoDB [23]. We use `local` Read Concern, which is the default setting, in all our experiments.

2.3 Data staleness in MongoDB

Let's first look at how MongoDB performs a write operation. When a write request reaches the primary copy of MongoDB, there will be an atomic transaction issued doing two things: 1) applying the database operation to the primary; 2) recording operations on the primary's operation log (called `oplog`). Once that transaction commits on the primary (and after waiting for any other transactions with earlier `oplog` entries to also commit), the `oplog` entry is visible to secondaries, and it will then be pulled by the secondaries and written to their `oplog`. After that the secondaries will apply the operations. Each node records the timestamp of the latest `oplog` applied (called `lastAppliedOpTime`). The `lastAppliedOpTime` of each node are known by all others.

By calculating the difference between the `lastAppliedOpTime` of a secondary node and the `lastAppliedOpTime` of the primary node, we can estimate the data staleness of the secondary node. Since the `lastAppliedOpTime` of one node is known by all other nodes, the comparison can take place at any of the MongoDB copies. This information can be provided to a client in the `serverStatus` command.

There is a potential source of error here. The `lastAppliedOpTime` for a secondary copy as recorded on the primary copy, might be earlier than the truth. This is because that the latest `lastAppliedOpTime` of the secondary copies might not yet have been transmitted to the primary copy. So the data staleness of the secondary, when calculated using statistics reported by the primary copy, might be larger than reality. Similarly, the `lastAppliedOpTime` of the primary copy as known on some secondary copy may be earlier than the current truth. So data staleness as calculated from the status at a secondary node can be smaller than reality. In Decongestant, we use the `serverStatus` information from the primary for this calculation, to be conservative in enforcing a client-requested limit on staleness.

3 DESIGN OF DECONGESTANT

In this section, we describe the design of Decongestant, an automated system to direct the reads dynamically for MongoDB during run-time of the application.

Our design needs to overcome some challenges. The foremost challenge is to have a single mechanism that can detect a variety of congestion situations. The bottleneck resource of a node of MongoDB, when it saturates, varies with different DBMS's hardware, configurations, and workload. It may be CPU usage, memory usage, even having a large number of Write Ahead Logs (WAL) waiting to be flushed to the disk. We also need a mechanism to detect when a secondary is too stale to be usable, considering the client's freshness requirement. An important challenge is that our mechanisms need to use only information readily available to the consumers of the MongoDB, as we do not do any modification of the server. On the client-side, we can observe requests and responses for the operations submitted to the service, and we also have some limited access to server information through available status reports (but this is much less detailed than what a server-side design could exploit).

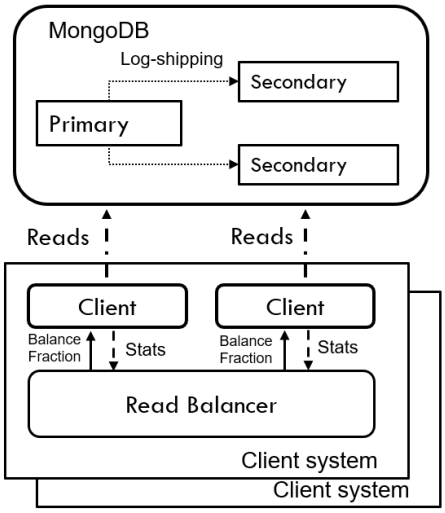


Figure 1: A simplified architecture of Decongestant

In this section, we first present an overview of Decongestant and we briefly introduce the decision-making component of the system, called Read Balancer. Then, we describe how the client application works in Decongestant. After that, we depict how Read Balancer does its job. Finally, we discuss some of the design details and considerations of Decongestant.

3.1 System Architecture

Figure 1 shows a simplified architecture of Decongestant. A component called Read Balancer resides on each client system where (maybe multiple) client applications are executing. The Read Balancer decides the percentage of read operations to be sent to the secondary copies. This percentage is to be chosen so as to improve the overall performance, by redirecting reads when one of the servers is congested, but not using secondaries at all if their data would be too stale to meet the client's specified demand for data freshness. The Read Balancer communicates with the clients via a few shared variables:

- The latest decision for each client on what percentage of the read-only transaction should be directed to the secondary copies, called *Balance Fraction*. The range of the Balance Fraction is between 10% to 90%, inclusive; or 0%. Balance Fraction is set to 0 when Decongestant estimates some secondary's data staleness exceeds the client's limit, so they will stop sending any read requests to the secondaries till this is remedied. The clients can express that they are not willing to accept any stale data, by setting the data staleness limit to 0.
- Two lists which keep track of client-observed transaction latencies (for reads that were sent to Primary and to Secondaries, respectively).

3.2 How do the client applications work in Decongestant?

In Decongestant, a client is expected to cooperate with the Read Balancer, as follows. Before invoking any read-only transaction call, the client first examines the most recent decision on the Balance Fraction from the Read Balancer. Then the client should flip a biased coin, and with a probability equal to the Balance Fraction, the call should to be directed the secondary copies of MongoDB. The clients keep track of the latency of their read-only transactions, and report them to the Read Balancer through one of the shared queues (depending on which Read Preference was actually used in the call).

3.3 How does the Read Balancer work?

In this section, we show how the Read Balancer acts. We first provide a high level overview of the Read Balancer and how it changes the Balance Fraction. Then, we discuss some technical details and considerations. Algorithm 1 shows the pseudo code for Read Balancer.

The Read Balancer periodically updates suggestions on the Balance Fraction, i.e. the percentage of read-only transactions that should go to the secondary copies. On system start up, the Read Balancer initializes the Balance Fraction as 10%. The Balance Fraction will be updated periodically (every 10 seconds in our implementation). We keep the Balance Fraction so it is never 0, except when this is needed because a secondary is too stale to use given the client-defined limit, or when the clients are not willing to accept any stale data. Similarly, Balance Fraction is never 100%. The reason to exclude the extreme values is to ensure that some reads are regularly using each server, so that the system has up-to-date data on the situation of the servers.

Within each period (when there is no need for the Read Balancer to recalculate its recommended fraction), the Read Balancer pings all nodes of the MongoDB cluster regularly, in order to record the Round Trip Time (RTT) between the client system and each node. In addition, the Read Balancer queries the primary node of the MongoDB replica set once per second, using the `serverStatus` command to get the latest data staleness estimation of each secondary node. Whenever the data staleness estimation on *any* secondary node exceeds the maximum data staleness the clients are willing to except, the recommended Balance Fraction will become 0, and so all read requests will be directed to the primary, until the data staleness situation on secondary nodes improves. (See detailed discussions in section 4.)

By the end of each period, the Read Balancer retrieves, from the shared lists, the recorded latencies of those read-only transactions that were sent to the primary during the period, and it calculates the median ("P50") value of these latencies; similarly it takes the list of latencies for read-only transactions sent to the secondaries, and determines the median of those. The Read Balancer then calculates a value called Server-Side Latency, for the reads on primary and secondaries, respectively. Server-Side Latency for a corresponding Read Preference option equals the median ("P50") latency of the read requests with that Read Preference option minus the median ("P50") Round Trip Time (RTT) of all MongoDB nodes corresponding to that the Read Preference option. See subsubsection 3.3.1 for details. These numbers act as estimates for the delay experienced by operations as they are performed on the server, including time spent within executing the MongoDB instance and also delays in the operating system or disk. The intuition here is that when a node is congested, this

Algorithm 1: Algorithm for Read Balancer

SharedVars: StaleBound - Client-set limit on data staleness
$L_{client,primary}$ - list of client-observed latencies of primary-sent ops
$L_{client,secondary}$ - list of client-observed latencies of secondary-sent ops
Bal - current Balance Fraction; initially LOWBAL

PrivateVars: RecentBal - list of 4 recent periods' Balance Fractions; initially all LOWBAL
Staleness - array of staleness reported by primary for each secondary
RTT - array of round-trip times to each server

Constants: DELTA - one-period change in Balance Fraction (10%)
LOWBAL - lowest value for non-zero Balance Fraction (10%)
HIGHBAL - highest value for Balance Fraction (90%)
LOWRATIO - latency ratio above which we increase Balance Fraction (0.75)
HIGHRATIO - latency ratio below which we decrease Balance Fraction (1.3)

```
1  Function Rcv-ServerStatus():
2      Update Staleness from ServerStatus
3      if (StaleBound == 0) or (max(Staleness) > StaleBound) then
4          Bal ← 0
5      else
6          Bal ← RecentBal.latest()
7      end
8  end
9  Function OnPeriodEnd():
10     L_ss,primary ← P50(L_client,primary) - P50(RTTprimary)
11     L_ss,secondary ← P50(L_client,secondary) - P50(RTTsecondary)
12     Ratio ← L_ss,primary / L_ss,secondary
13     if Ratio > HIGHRATIO then
14         NewBal ← min(RecentBal.latest() + DELTA, HIGHBAL)
15     else if Ratio < LOWRATIO then
16         NewBal ← max(RecentBal.latest() - DELTA, LOWBAL)
17     else if All RecentBal entries are the same then
18         NewBal ← max(RecentBal.latest() - DELTA, LOWBAL)
19     else
20         NewBal ← RecentBal.latest()
21     end
22     RecentBal.dequeue().enqueue(NewBal)
23     if (StaleBound == 0) or (max(Staleness) > StaleBound) then
24         Bal ← 0
25     else
26         Bal ← RecentBal.latest()
27     end
28 end
```

figure will increase, no matter whether the bottleneck resource is in MongoDB or elsewhere on the server.

The Read Balancer uses the ratio of the Server-Side Latency of reads on the primary, to the Server-Side Latency of reads on the secondary. To achieve the maximum performance possible of the MongoDB Cluster, all the nodes in the cluster should do the heavy-lifting together, sharing the work. This means, ideally, the ratio should be close to 1. If the ratio is much larger than 1, it means the primary copy is congested compared to the secondary copies. In this case, the Read Balancer increases the Balance Fraction for the next period, directing more reads to the secondary nodes. On the other hand, if the ratio of Server-Side Latency is much less than 1, indicating that the secondary nodes are congested more than the primary nodes, then the Read Balancer would decrease the Balance Fraction, sending less reads to the secondary copies in the next period. If the ratio is close to 1 and it has been close to 1 for quite a while, Read Balancer would also decrease the Balance Fraction to explore "downward". This is to make sure the reads go to the primary node as much as possible, in order to improve the data freshness and avoid potential stale reads [26]. Unlike our previous work [21], the Read Balancer does not look for one "correct" Read Preference in each period. Instead, it tries to balance the workload among all the nodes to achieve the best performance possible.

3.3.1 Server-Side Latency. As discussed before, the Read Balancer utilises Server-Side Latency to make decisions. Server-Side Latency of a Read Preference option is found as follows: we take the median latency of all read-only invocations with that Read Preference as observed on the client side in the previous period, minus the median Round Trip Time (RTT) of all MongoDB nodes corresponding to that Read Preference choice.

$$L_{ss} = P_{50}(L_{client}) - P_{50}(RTT)$$

In MongoDB, all nodes are spread across different availability zones within a region as much as possible, when deployed in a public cloud. The Round Trip Time (RTT) between a certain client and nodes in different availability zone varies. Although the difference is usually less than 2 milliseconds, it is enough to impact the latency observed on the client side for those workloads with light read-only transactions, such as YCSB, where the latencies themselves are sometimes only 1 to 2 milliseconds.

3.3.2 Bounded Data Staleness. The Read Balancer also talks to the primary node of the MongoDB replica set several times each period, to call `serverStatus` at the primary. It uses this information as described in subsection 2.3 to conservatively estimate the data staleness of each secondary node.

The clients can tell the Read Balancer the maximum data staleness they are happy to accept. As we have explained before, the MongoDB clients can only choose to read from either the primary node or from any secondary node, they can not specify which secondary node they would like the read requests to be sent to. So, as long as the Read Balancer finds the estimated data staleness of any secondary node is larger than the threshold set by the clients, the Read Balancer immediately notifies every client, that all future read requests should only be sent to the primary copy - the Balance Fraction is set to be zero. The Read Balancer resumes a non-zero Balance Fraction once the maximum estimated data staleness of the secondary copies drops below the threshold set by the clients.

One concern may be raised here. The Read Balancer restarts sending read requests to the secondary nodes once the maximum data staleness of all secondary nodes drops below the threshold. Would that extra work cause the data staleness to quickly go beyond the limit again? We had the same concern. Our empirical study shows that in MongoDB, data staleness increases gradually on the secondaries, but when it goes down, it drops swiftly, to nearly zero. The high-level idea is that the gradually increasing data staleness is caused by a congested primary node, which is too busy processing data requests so it is not able to provide the `oplog` to the secondary copies. But once the `oplog` is sent, the secondary nodes catch up quickly. See Section 4.5.

The data staleness for secondaries which is estimated by this method may be larger than what a client actually observes. As well as the possible overestimate from the primary copy not yet knowing very recent activity updating the secondary, there is also the possibility that there are `oplog` entries not yet applied to the secondary copies but these might not modify the particular data the MongoDB client queries. We are conservative, and avoid using a secondary if our (perhaps over-) estimate breaches the client-set limit on staleness. Section 4.5 reports experiments to check the alignment between the data staleness estimate used in our decision, and measurements in a targeted S-workload that observes latency at clients.

4 EVALUATION

In the following, we present an evaluation of Decongestant. We first introduce the methodology and settings we use in Section 4.1. We then demonstrate Decongestant's ability to detect and adapt to variation of workloads in Section 4.2. We explore Decongestant's ability of balancing the load among all MongoDB nodes to achieve better performance than using current practice in a read-intensive workload (Section 4.3), and Decongestant's capability of trading data freshness for performance in the workloads with a mixture of reads and writes (Section 4.4). Section 4.5 covers Decongestant's competence of bounding the data staleness. Finally, we show that running S workload concurrently has low impact on performance measurements of standard workloads, in Section 4.6.

4.1 Method

4.1.1 Platform. The experiments are executed on AWS. The MongoDB clients are on an AWS c4.4xlarge instance (16 vCPUs and 30 GB RAM), located in the region ap-southeast-2.

We deploy our own MongoDB cluster on AWS, in order to maintain consistent results independent of infrastructure and software version changes that are out of our control in MongoDB Atlas. We replicate the configurations from MongoDB Atlas in June 2020. The MongoDB version is 4.2.6. A 3-node MongoDB cluster is deployed on 3 AWS r4.2xlarge instances, which has 8 vCPUs and 61 GB RAM, located in the same region as the clients but different Availability Zones: ap-southeast-2a, ap-southeast-2b, and ap-southeast-2c, respectively.

4.1.2 Decongestant settings. In all following experiments, Decongestant considers the ratio of Server-Side Latency of the reads on the primary to the Server-Side Latency of the reads on the secondaries as being normal when the value ranges from 0.75 to 1.30. When the ratio is greater than 1.30, Decongestant considers that the primary is congested, and thus it increases the percentage of reads sent to the secondaries in the next period by 10%. A ratio less than 0.75 leads Decongestant to send 10% fewer reads to the secondary nodes in the next period, as it shows that the secondaries are more congested. Balance Fraction starts at 10%

Table 1: Percentage of transactions in the original TPC-C workload versus the read-write TPC-C workload used in the experiments.

	TPC-C	Read-Write TPC-C
Stock Level	4%	**50%**
Delivery	4%	4%
Order Status	4%	4%
Payment	43%	**20%**
New Order	45%	**22%**

and is capped at 90%. Decongestant revisits the Balance Fraction decision every 10 seconds. In our experiments, unless stated explicitly, the maximum data staleness the clients are willing to accept is set to be 10 seconds.

Read Balancer keeps the four previous records of the Balance Fraction. If they remains the same, Read Balancer pushes down the Balance Fraction by 10% in the next round.

4.1.3 Baselines. As well as showing the performance of our prototype system Decongestant, we also look at two baselines corresponding to current practice. In these, the clients run against MongoDB, without any Read Balancer installed, or any of the possible overheads of communicating with it; in the baseline Primary, each read is hard-coded with Read Preference set to `primary`. In baseline Secondary, each read is hard-coded with Read Preference as `secondary`.

4.1.4 Workloads. Two different sets of transactions are used here to evaluate the performance of Decongestant. We use YCSB [12] with varying read-write percentage: YCSB-A (50% reads and 50% writes) and YCSB-B (95% reads and 5% writes). YCSB represents a light-weight workload, with simple get and put operations. As a more demanding workload, we also use a variant of TPC-C (we call it read-write TPC-C). This uses the transactions from TPC-C, but unlike write-heavy traditional TPC-C, we aim here for a balance between read-only and update transactions. To do so, the percentage of Stock Level transaction, which represents a read-only transaction, is set to 50%. Table 1 shows the detailed breakdown of the read-write TPC-C workload as compared to standard TPC-C. The TPC-C queries are implemented by Kamsky [29] who has adapted the TPC-C benchmark to the MongoDB query language and transaction semantics, as well as adapting it to MongoDB's best practices.

4.1.5 S workload to monitor data staleness. In order to check whether Decongestant keeps the maximum data staleness promise, we need a method to sample the data staleness from the client side. While Decongestant uses estimates for staleness based on MongoDB's internal information on the status of oplog application, we want to validate our system's success using real measurements. So, we propose S workload (S stands for staleness). The S workload could be run standalone, but, in all our experiments[2], we generate this S workload alongside the main performance-focused OLTP workloads, such as YCSB, TPC-C.

The high-level idea of S workload is similar to our previous work [20, 40]. The S workload includes two workers (each worker can be implemented as a separate process or a thread): one writer and one reader. The job of the writer is to keep writing the current timestamp to a dedicated item in the database at a high frequency. It is not necessarily to write as fast as possible, but it should work at least as fast as the reader does.

Periodically, the reader probes the contents from the various copies of the dedicated cell. In each probe, the reader sends out two read requests: one with the Read Preference Primary and the other with the Read Preference Secondary. The reader records the results. Then, in the analysis phase (after the experiment), we can determine the freshness of the data returned from the secondary reads by comparing the results between the primary read and the secondary read. A slight variation is done at times when the main application is not using the secondaries at all (and so clients will see no staleness at these times): the second read in each probe of S workload can be simply directed to the primary copy again.

The staleness monitoring approach used by S workload is a bit different from the one used by [20, 40]. In those prior works, the reader only sends one read request in each probe, with the Read Preference Secondary; during the analysis phase, the timestamp when the read is sent and the read value are compared, to determine the staleness gap. The assumption in their approach was that the timestamp in the dedicated cell in the object database keeps advancing smoothly. This assumption doesn't always hold when monitoring is run together with another existing OLTP workload. There are times when a write takes a long time to finish, causing the value in the dedicated cell of the primary copy to be unchanged for a while. By definition, the value in the primary copy is fresh. However, the timestamp when the read request is sent keeps going forward as the S-reads are generated continuously. So, if the approach described in [20, 40] is used, fake staleness might be reported.

Note that the data staleness measured by this S workload might be larger than the data staleness seen by any given application client, as S workload frequently updates the item being read, while some application read may be on slowly-changing data, where the returned value is correct even if the secondary is far behind in applying the oplog. Still, if we succeed in bounding staleness seen in S reads, we can be sure that any other application also has data that is at least this fresh.

4.1.6 Measurements. The experiments have two distinct styles. In some, we report on the time-varying properties through a run of a system when a workload is applied (perhaps the workload changes at specific points during the run). In those experiments, we report for each separate 10 second period on throughput of appropriate transactions during the period, P_{80} latency (that is, the time within which were completed 80% of reads of the period), the percentage of reads that were sent with Read Preference Secondary during the period, and sometimes data staleness, either as measured by S workload, or as estimated conservatively using reports from the primary of the max difference between any secondary's `lastAppliedOpTime` and that of the primary. Although Server-Side Latency is used by Read Balancer to make decisions, all latency reported in the following figures are end-to-end latency observed by the clients. Measuring the actual percent of reads sent to the secondary copies is done by counting the read operations which were sent to the primary copy and those sent to secondary nodes; we do not simply echo what Decongestant suggests through Balance Fraction.

Other experiments give a single data point for overall performance (throughput, latency or staleness) for runs in some situation, typically shown in a figure where some important parameter of the workload (eg number of clients) varies along the

[2] except in one experiment of Section 4.6.

x-axis. In these experiments, except where stated explicitly, each data point is taken from the average over 3 runs; in each run, measurements exclude the first 100 seconds, which is treated as a warm-up period.

4.2 Adapting to dynamic workload

We first claim that Decongestant is able to adapt well to variation of workloads. We compare it to the outcomes for the two baseline systems, where all reads have hard-coded read preference setting, so either all go to the primary, or all to the secondaries.

Figure 2 shows the throughput of read operations, 80-percentile latency (end-to-end) of read operations, and the actual percentage of reads which are sent to the secondary copies, during a run with a dynamically-changing workload. The workload in this run starts with YCSB-A (50% reads) with 180 clients, and swaps to YCSB-B (95% reads) at the 620^{th} second. S workload is running as well, throughout. For the first 90 seconds, Decongestant warms up shifting from its initial setting with 10% of reads on secondaries, over time sending more and more read operations to the secondaries (Figure 2 (c)), until the highest amount we allow (90%) of the reads are secondary ones. During this period, the throughput increases (Figure 2 (a)) and the 80-percentile latency drops (Figure 2 (b)). From the 90^{th} second to the 620^{th} second, the percentage of reads sent to the secondary copies stabilises at 90%. This is the same performance we achieved in previous work [21]. At the 620^{th} second, the workload shifts from YCSB-A (50% reads) to a read-dominated YCSB-B (95% reads). Decongestant quickly responds by sending less reads to the secondary. The percentage of reads routed to the secondary nodes stables at 70% under this workload. The intuition is that the primary node deals with 5% writes and a bit less than 1/3 of reads, and two secondary nodes process between them a bit more than 2/3 of the reads. Recall that our MongoDB cluster is a three-node cluster with the same capacity in each node. This shows that Decongestant successfully balances the read load proportionally to the capacity behind each Read Preference option. As a result, the throughput (Figure 2 (a)) of the read operations is higher than either baseline (only sending read requests to the primary or secondaries); and the 80-percentile (Figure 2 (b)) latency is better than baselines.

Once the system has adjusted to the changed workload, the ratio of Server-Side Latency of the reads on the primary to the Server-Side Latency of the reads on the secondaries remains between 0.75 to 1.30. During this period, Read Balancer tries to push more reads to the primary on every fifth period. But as the Read Balancer finds it does not work well, the Read Balancer bring the Balance Fraction back to 70%.

Figure 3 is another example showing that Decongestant successfully notices and adjusts to the workload shifts; here both read-intensity and total load change. At the beginning the workload is YCSB-B (95% reads) with 180 clients. Then, after 230 seconds, it shifts to YCSB-A with 20 clients. On system start up, Decongestant increases the percentage of reads sent to the secondaries (Figure 3 (c)). The percentage soon reaches an optimised state at 70%. During this period, the throughput of read operations in Decongestant is higher (Figure 3 (a)) and the 80-percentile latency of those reads (Figure 3 (b)) is lower than when the Read Preference is hard-coded as Primary or Secondary. At the 230^{th} second, the workload switches to YCSB-A (50% reads) with 20 clients. Decongestant quickly decreases the percentage of reads sent to the secondary, as the primary can now handle all the load. The allocation becomes stable at the minimum we allow (10%), to make sure Read Balancer keeps getting enough recent information on the state of the secondaries, so we will detect future congestion if it were to happen.

Figure 4 shows the performance of Decongestant with the dynamic read-write TPC-C workload. Unlike previous experiments on YCSB, the throughput and 80-percentile latency (end-to-end) here are reported for each 1 minute interval, and the actual percentage of secondary Stock Level transactions are recorded every 10 seconds. The workload starts with 20 clients, and then at the 5^{th} minutes the client number increases to 200. After 5 more minutes, it goes down to 20 clients. Figure 4 (c) shows the actual percentage of secondary Stock Level transactions. It starts at around 10%. From the 5^{th} minute, Read Balancer is able to notice the high contention environment, and quickly pushes up the percentage of secondary Stock Level transactions. This soon brings the throughput (Figure 4 (a)) and 80-percentile latency (Figure 4 (b)) of Decongestant to a level similar to the situation where the Read Preference is "hard-coded" as Secondary. During the 5^{th} minute to the 10^{th} minute, there are a few downward spikes in the actual percentage of secondary Stock Level transactions. They are caused when the maximum data staleness on the secondary copies exceeds the clients' threshold, which is 10 seconds; this is detected by Read Balancer, so we stop sending Stock Level transactions to the Secondary copies. The measured percentage is not 0, as the staleness check is run once per second, while the percentage reported here is taken over 10 seconds. The pink vertical lines in the figure shows the seconds where all reads are directed to the primary, due to exceeding data staleness. After the 10^{th} minute, the number of clients drops to 20. Read Balancer gradually brings back most of the Stock Level transactions to the now-uncongested primary node, to provide lower staleness.

4.3 Achieving better performance by sharing load in read intensive workloads

In this section, we show how Decongestant's capability of balancing the read load among *all* the nodes of MongoDB cluster, achieves a better peak performance for read-intensive workloads, compared to hard-coding Read Preference as either Primary or Secondary. Each hard-code approach leaves some node underloaded. Decongestant does not try to specially identify whether the workload is read intensive; instead, the exact same approach is used throughout. Each data point in a plot is the average over three runs, excluding the warm-up period.

Figure 5 shows the throughput of reads, 80-percentile latency (end-to-end) of reads, and the actual percentage of reads sent to the secondary copies, plotted against number of clients, in YCSB-B (95% reads). We first discuss the actual percentage of reads sent to the secondary copies (Figure 5 (c)) over a varying number of clients. We can see with a low load of between 10 to 50 clients, Decongestant sends most read requests to the primary node. Decongestant sends more reads to the secondary copies, with the percentage growing corresponding to client number, in the range between 50 to 100 clients. The percentage of secondary reads is roughly stable at around 70% when the number of clients ranges from 120 to 200. This means, in a MongoDB cluster with 3 nodes of the same capacity, the primary nodes deals with 5% write operations and around 30% read operations; while the two secondary nodes between them process 70% read requests. This shows that all three nodes do the heavy-lifting together.

When the client number is between 120 to 200, all three nodes work together, instead of the primary node or the secondary

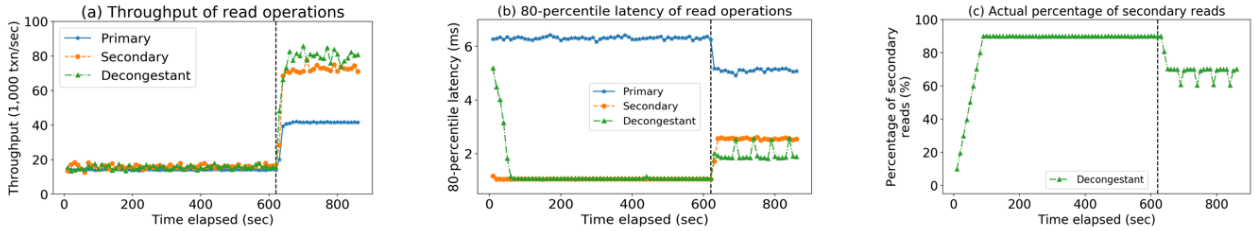


Figure 2: Decongestant's ability to respond to sudden increase of the read-write ratio in YCSB. The plots show the throughput of read operations, 80-percentile latency (end-to-end) of read operations, and the actual percentage of reads sent to the secondary copies, in a dynamic workload. The dynamic workload starts with YCSB-A (50% reads) with 180 clients, and swaps to YCSB-B (95% reads) at the 620th second. The vertical dotted line shows the time when the variation happens.

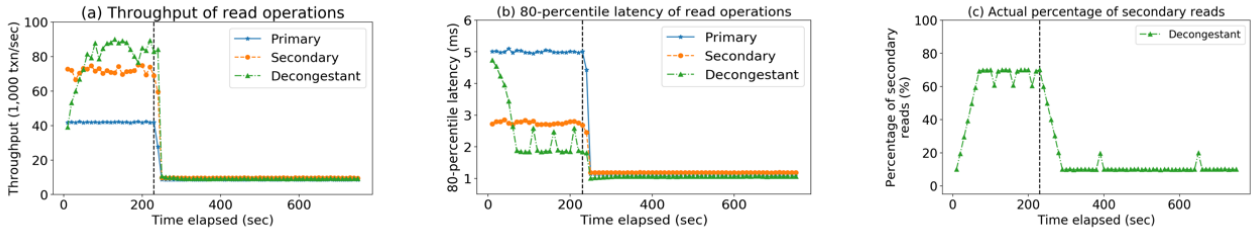


Figure 3: Decongestant's ability to respond to sudden decrease of both the read-write ratio and the number of clients, at the same time, in YCSB. The plots show the throughput of reads, 80-percentile latency (end-to-end) of reads, and the actual percentage of reads sent to the secondary copies, in a dynamic workload. The dynamic workload starts with YCSB-B (95% reads) with 180 clients, and swaps to YCSB-A (50% reads) with 20 clients at the 230th second (indicated by the vertical dotted line).

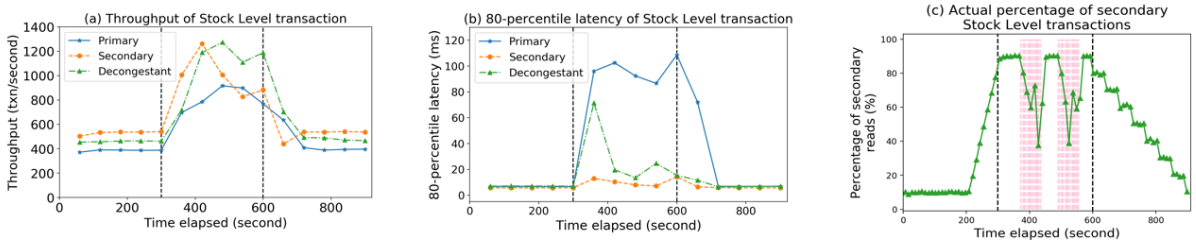


Figure 4: Decongestant's response to variations of client number in read-write TPC-C. The client number starts at 20. It bursts to 200 clients at the 5th minute. The client number drops back to 20 at the 10th minute. The black vertical dotted lines show the times when the variations of the client number happen. The plots display the throughput of Stock Level transactions, their 80-percentile latency (end-to-end), and the actual percentage of Stock Level transactions sent to the secondary copies. The throughput and the 80-percentile latency (end-to-end) are reported on per-minute bases, while the actual percentage of secondary Stock Level transactions are recorded once per 10 seconds. The pink vertical dotted line shows the seconds where all reads are directed to the primary, due to exceeding data staleness.

nodes only, so it is not surprising that Decongestant is able to achieve a throughput (Figure 5 (a)) that is around 30% higher than only routing the read requests to the secondary copies, and around 2.5 times than only sending read operations to the primary node. We also see that the 80-percentile latency for Decongestant is lower that the two hard-coded systems when the load is high.

We do not plot data staleness for YCSB-B (95% reads) here. The 80-percentile data staleness, measured both by S workload and Decongestant, for YCSB-B are constantly zero in all our experiments. This situation is not very surprising, as there are only 5% of writes in the workload. There are occasionally one or two data staleness values observed to be one second by the S workload, when the Read Preference is "hard-coded" as Secondary. Since the granularity for the data staleness is one second, we do not feel it is very meaningful.

4.4 Trading data freshness for performance

In this section, we show that Decongestant is able to trade data freshness for performance when needed in the workloads with a mixture of reads and writes. Again, we point out that Decongestant never tries to identify whether a workload has a mixture of

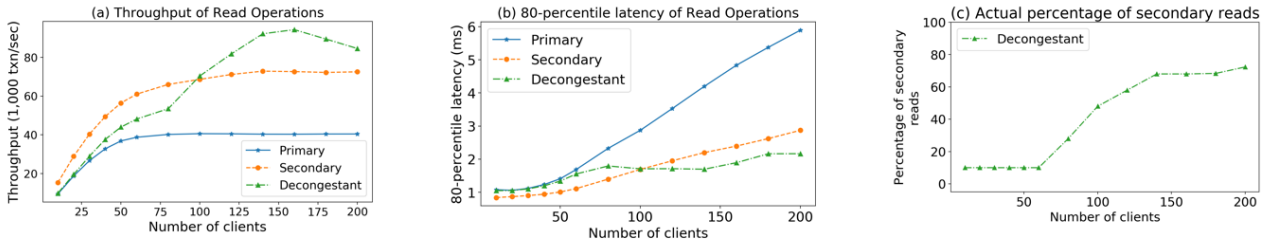


Figure 5: Performance trends with increasing client count. The plots show the throughput of reads, their 80-percentile latency (end-to-end), and actual percentage of reads sent to the secondary copies, against the number of clients in YCSB-B (95% reads).

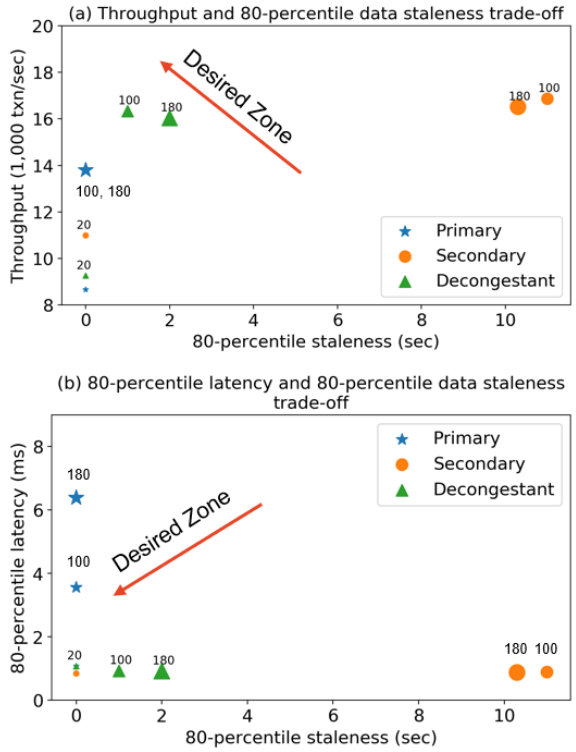


Figure 6: Performance and 80-percentile data staleness trade-off in YCSB-A for read operations. (a) shows the trade-off between the throughput and the 80-percentile data staleness of the read requests. The number above the marks and the size of them indicates the number of clients. The upper left corner is the desired area for Decongestant, with large throughput and small data staleness. (b) shows the trade-off between 80-percentile latency (end-to-end) and 80-percentile data staleness for read operations. The lower left corner is the target zone for Decongestant, with small latency and data staleness.

reads and writes or it is a read-intensive one: the same method works for either scenario. The experiments in this section are each run three times for each setting, and the average value is taken and shown as a point in the charts.

Figure 6 shows the performance and the data staleness trade-off for YCSB-A (50% reads). To avoid the figures being too cluttered, we only include 3 groups of data points here. The number above the marks and the size of them indicates the number of clients. Three client numbers are chosen: 20, 100, and 180, representing light, medium, and heavy load, respectively. Figure 6 (a) demonstrates the trade-off between the throughput of reads and the 80-percentile data staleness of them. The upper left corner is the desired zone in Figure 6 (a), with small data staleness and large throughput. When the load is low (20 clients), the data points for the three situations, whether always directed to either the primary or secondary nodes or with Decongestant, are close. When the load is medium (100 clients) and large (180 clients), Decongestant is able to push the data point close to the desired zone while the baselines are far away (Primary having low throughput, and Secondary seeing high staleness).

Similar conclusions can be reached for the trade-off between latency and data staleness from Figure 6 (b). The lower left corner is now the ideal area, as it shows small values in both latency and data staleness. The advantage of using Decongestant is clear, offering a sweet spot in terms of 80-percentile latency and 80-percentile staleness.

Figure 7 shows the performance and the data staleness trade-off in a read-write TPC-C workload (50% reads). Figure 7 (a) reports the trade-off between the throughput of the Stock Level transaction and the data staleness, where the upper left corner is desirable. Figure 7 (b) depicts the trade-off between latency of Stock Level transactions and data staleness, where the lower left corner is the target. Decongestant is able to push the data points toward the desired zone.

4.5 Bounding data staleness

In this section, we discuss the data staleness issue. We make two claims here:

- Decongestant's estimation of data staleness, from `serverStatus` reports, closely aligns with the data staleness seen by the clients.
- When the maximum data staleness of a secondaries copy exceeds the threshold set by the clients, the clients of Decongestant will not see this.

The largest data staleness the clients are willing to accept is set to 10 seconds, and we measure the staleness seen by clients, with S workload from Section 4.1.5.

Figure 8 compares the data staleness from `serverStatus` to the one seen by the clients, against time elapsed. The workload is one run of YCSB-A together with S workload, with. 100 clients. This figure shows that maximal data staleness known by the Decongestant via MongoDB `serverStatus` (and used by Decongestant to detect cases where excessive staleness means that reads should avoid the secondary nodes), aligns quite well

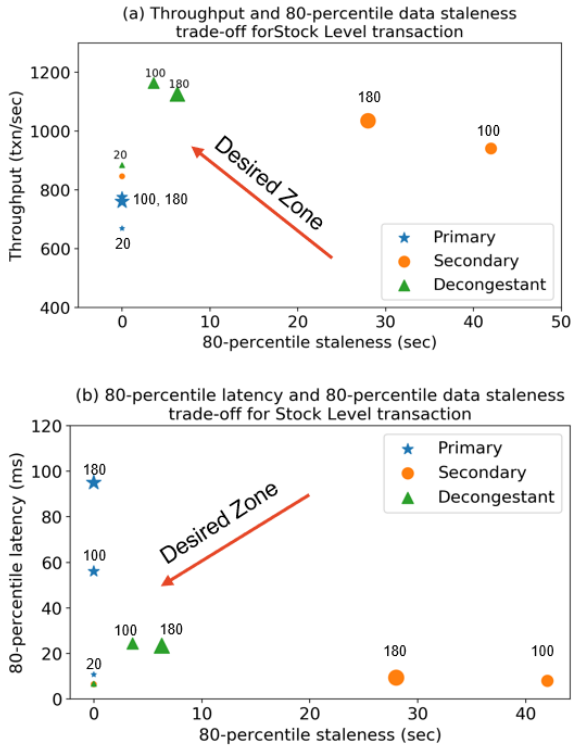


Figure 7: Performance and 80-percentile data staleness trade-off for Stock Level transacts with read-write TPC-C. (a) shows the trade-off between throughput of Stock Level transacts and 80-percentile data staleness. The number above the marks and the size indicates the number of clients. The upper left corner is the desired area for Decongestant with both large throughput and small data staleness. (b) shows the trade-off between 80-percentile latency (end-to-end) of Stock Level transactions, and 80-percentile data staleness. The lower left corner is the target zone for Decongestant with both small latency and data staleness.

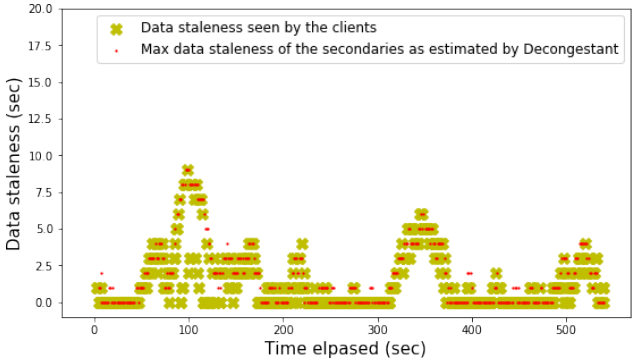


Figure 8: The maximum data staleness of the secondaries estimated by the Decongestant versus the one seen by the clients, against time elapsed. The workload is YCSB-A together run with S workload, from 100 clients. The data staleness estimated by Decongestant is good as long as it is not smaller than the one seen by the clients.

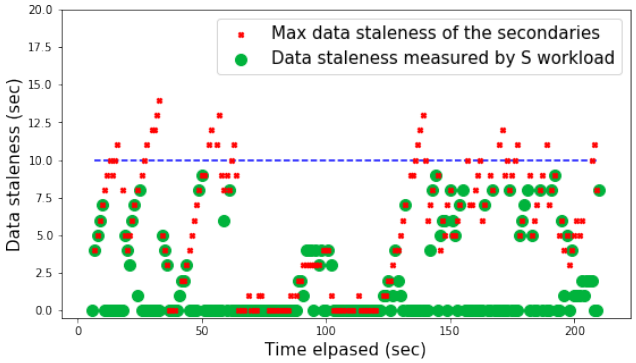


Figure 9: Data staleness measured by S workload versus the max data staleness of the secondaries in Decongestant, with the data staleness limit set to be 10 seconds. The workload used here is read-write TPC-C with 60 clients.

with the data staleness seen by the clients. In some cases, we can see that the data staleness estimated by the Decongestant is larger than the ones seen by the clients. This is acceptable, as the Decongestant records the maximum data staleness among all secondaries, while the S workload measures the data staleness on an arbitrary secondary node.

Figure 9 shows that when data staleness of a secondary copy exceeds the threshold set by the clients, the clients of Decongestant will not see this: Decongestant reacts in time and sends their reads to the primary. The workload used here is read-write TPC-C with S workload, on 60 clients. The blue horizontal dashed line shows the data staleness limit set by the clients, which is 10 seconds. The red squares in the figure depict the maximum data staleness of the secondaries, which sometimes goes beyond the threshold. However, the Decongestant clients are protected from this — all green circles are below their data staleness limit.

We have done a more detailed analysis on the MongoDB internal diagnostic data, trying to understand what happens behind this data staleness pattern (Figure 9). The high-level idea is that when the primary is overloaded, the secondaries trying to read the oplog from the primary can stall. That is, a secondary has a cursor open on the primary's oplog and calls the "getMore" operation. When the primary is overloaded, it can take a long time to service this operation, and during that time, the secondary can't move forward because it does not know about the newer operations (i.e., it gets more and more stale). Eventually the primary gets around to servicing the "getMore" request and sends a large batch of operations to the secondary. Since the secondary isn't overloaded, it can apply the operations quickly and catch up.

Now we discuss how the primary is stalled. During this period, several checkpoints completed on the primary and the disk of it was 100% utilised. The checkpoints took a long time to complete (around 30 seconds on average). During that time the latency grows. MongoDB had noticed the lag and was deliberately throttling update operations via a mechanism called "flow control" [27]. This might be part of the reason for the throughput of read-write TPC-C workload being unstable (Figure 4 (a)). Once the flush completes, the primary comes back to life and serves a batch of "getMore" operations quickly.

Our method to bound the data staleness still works reasonably well when the clients set the data staleness limit to a very low value, such as 3 seconds (see Figure 10). Such a low data staleness limit is challenging, as the granularity of the data staleness reported by MongoDB `serverStatus` is one second. So a system has little time to react to increasing staleness before the limit

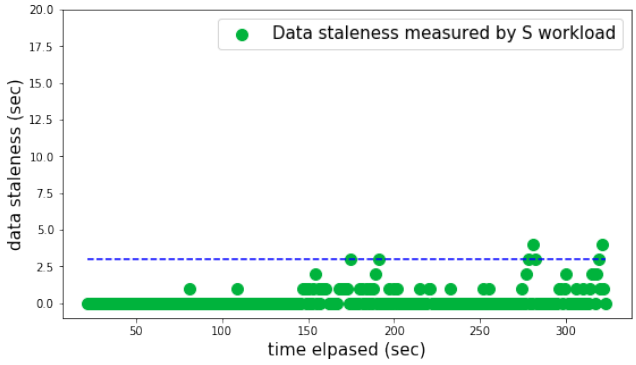


Figure 10: Data staleness measured by S workload in Decongestant with the data staleness limit set to be 3 seconds. The workload used here is read-write TPC-C with 200 clients.

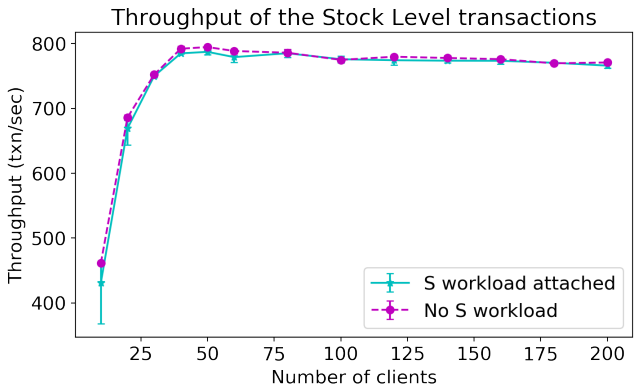


Figure 11: Throughput of Stock Level transactions when running the read-write TPC-C workload, with or without S workload.

is breached. The workload used in Figure 10 is read-write TPC-C with 200 clients. We can see, in most cases, the client-observed data staleness are bounded as requested, though two data points have staleness value 4 seconds.

4.6 Impact of S workload

Lastly, we claim that running S workload alongside a performance benchmark (such as read-write TPC-C), causes little distortion on the results recorded for performance. Figure 11 shows the throughput of Stock Level transactions, when running the read-write TPC-C workload with the S workload (as it does in our previous experiments), compared to what is measured when the read-write TPC workload is run alone. The Read Preference used here is Primary. We can see that the throughput of the performance benchmark remains at a similar level when running with or without the S workload.

5 RELATED WORK

Some text in this section already appeared in [21].

The trade-off between performance and consistency in distributed storage system has been studied for a very long time. This body of work is profoundly influenced by CAP theorem [9] and its later PACELC formulation [1].

Various storage systems offer different consistency properties. Some make the choice for the users, such as BigTable [10] and Spanner [13], which guarantee some form of strong consistency. Others give users some freedom to make their own decisions, including Amazon Dynamo [16], Cassandra [30], as well as MongoDB. There is also a trend that Database-as-a-Service is hosted on top of shared-disks systems (usually provided by large cloud computing vendors), such as Amazon Aurora [39] and Microsoft Socrates [3], whose features usually impact on the exposed consistency characteristics of these Database-as-a-Service.

There is a huge amount of work evaluating the behaviour of distributed storage systems, in order to help users make more informed choices. Wada et al and Bermbach et al benchmark a large variety of distributed storage systems from the customers' view [5–8, 40]. These works treat the distributed storage system as a black-box, and send read and write requests to it, just like normal customers would do. Our previous work [20] measured the inconsistency window between the primary copy and secondary copies of MongoDB Atlas at around 25 ms.

There are works which, rather than comparing the read and write results (as client-centric methods do), capture the trace of operations on various entities (chosen by the user), and then post-execution analysis determines whether an equivalent serial execution would give the same result in each read; if not an anomaly is reported [2, 17, 31].

Trading performance for data freshness for read-only transactions / queries has been explored [11, 18, 33]. For example [33] applied this idea to mix OLAP and OLTP workloads in a database cluster providing freshness guarantees, though requiring a central coordinator which sees all transactions. In contrast, our proposed Decongestant is decentralised, which can be used both by on-premises and cloud users, and can deduce overload situations by probing rather than seeing the complete workload.

Pileus is a self-configuring system based on a Service Level Agreement (SLA) [36]. Within one SLA, there are a few subSLAs. Each subSLA includes a consistency requirement, a latency bound, and a utility score. Similar to our work, Pileus has "monitors" residing on the client nodes (each client has one monitor), and these probe periodically to decide which node a reading request should be directed to, so that the highest utility score among all subSLAs is achieved. Tuba [4] is an extension for Pileus. Tuba is a DBMS which is able to reconfigure itself, based on the observed latency and subSLA hit and miss ratio from all clients. Possible reconfiguration includes: changing primary replica, adding or removing secondaries, and varying synchronization periods between the primary and secondary copies.

Some recent work for self-configuring database applies machine learning technologies. There are automated systems able to tune large number of database knobs [38], adding and deleting indices [14], forecasting workloads [32], scaling resources [15], providing advice on partitioning [19], etc.

6 CONCLUSION

We presented the design and evaluation of Decongestant, a system which is able to automatically and dynamically determine Read Preference settings for read operations in MongoDB, in order to get good performance while delivering fresh-enough data to clients. Our solution works for both on-premises MongoDB deployments and MongoDB-as-a-Service. The key innovation is a client-based way to detect when either the primary, or one of the secondaries, are congested, by comparing estimates of the

time taken on the relevant server for performing recent read operations. When congestion is detected through these estimates, the system shifts reads away from the congested server; however, this decision is also constrained by estimates of the current data staleness on the secondaries.

This design avoids the need in current practice for application programmers to hard-code the decision of whether reads should go to primary or to secondaries (and thus risk seeing stale values). Instead the decision is made dynamically at run time by Decongestant, adapting to the recent situation in the servers.

Our experimental evaluation is done with YCSB-A, YCSB-B, and with workloads that run TPC-C transactions with a balance between read-only and updating transactions. We showed that Decongestant is able to adapt to workload shifts as they occur, and that it delivers good performance that respects client-chosen limits on data staleness. Indeed, in read-intensive workloads such as YCSB-B, we can outperform both hard-coded alternatives.

In future work, we plan to look at more sophisticated feedback control when adjusting read preference, and to support richer client SLAs as well as maximum staleness. We will explore the possibility of extending Decongestant to other database systems, which have a leader-follower architecture similar to MongoDB.

ACKNOWLEDGMENTS

This work was supported by the Australian Research Council (ARC) Linkage Project LP160100883.

REFERENCES

[1] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.

[2] Yazeed Alabdulkarim, Marwan Almaymoni, and Shahram Ghandeharizadeh. 2017. *Polygraph.* Technical Report 2017-02. Database Laboratory, Computer Science Department, University of Southern California.

[3] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'19)*. 1743–1756.

[4] Masoud Saeida Ardekani and Douglas B Terry. 2014. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 367–381.

[5] David Bermbach. 2014. *Benchmarking eventually consistent distributed storage systems.* KIT Scientific Publishing Karlsruhe.

[6] David Bermbach and Stefan Tai. 2011. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proceedings of 6th Workshop on Middleware for Service Oriented Computing*. ACM, 1.

[7] David Bermbach and Stefan Tai. 2014. Benchmarking eventual consistency: Lessons learned from long-term experimental studies. In *IEEE International Conference on Cloud Engineering (IC2E'14)*. 47–56.

[8] David Bermbach, Erik Wittern, and Stefan Tai. 2017. *Cloud service benchmarking*. Springer.

[9] Eric A Brewer. 2000. Towards robust distributed systems. In *PODC*. 7.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems* 26, 2 (2008), 4.

[11] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A.N. Soules, and Alistair Veitch. 2012. LazyBase: Trading Freshness for Performance in a Scalable Database. In *Proceedings of 7th ACM European Conference on Computer Systems (EuroSys'12)*. 169–182.

[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. 143.

[13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems* 31, 3 (2013), 8.

[14] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically indexing millions of databases in Microsoft Azure SQL database. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'19)*. 666–679.

[15] Sudipto Das, Feng Li, Vivek R Narasayya, and Arnd Christian König. 2016. Automated demand-driven resource scaling in relational database-as-a-service. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'16)*. 1923–1934.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007 (SOSP 2007)*. ACM, 205–220.

[17] Wojciech Golab, Muntasir Raihan Rahman, Alvin AuYoung, Kimberly Keeton, and Indranil Gupta. 2014. Client-centric benchmarking of eventual consistency for cloud storage systems. In *IEEE 34th International Conference on Distributed Computing Systems (ICDCS'14)*. 493–502.

[18] Hongfei Guo, Per-Åke Larson, and Raghu Ramakrishnan. 2005. Caching with 'Good Enough' Currency, Consistency, and Completeness. In *Proceedings of 31st International Conference on Very Large Data Bases (VLDB'05)*. 457–468.

[19] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2019. Towards learning a partitioning advisor with deep reinforcement learning. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. ACM, 6.

[20] Chenhao Huang, Michael Cahill, Alan Fekete, and Uwe Röhm. 2018. Data Consistency Properties of Document Store as a Service (DSaaS): Using MongoDB Atlas as an Example. In *Technology Conference on Performance Evaluation and Benchmarking (LNCS 11135)*. Springer, 126–139.

[21] Chenhao Huang, Michael Cahill, Alan Fekete, and Uwe Röhm. 2020. Deciding When to Trade Data Freshness for Performance in MongoDB-as-a-Service. In *IEEE 36th International Conference on Data Engineering (ICDE'20)*. 1934–1937.

[22] MongoDB Inc. 2020. MongoDB Documentation: Sharding. https://docs.mongodb.com/manual/sharding/. Accessed:2020-05-06.

[23] MongoDB Inc. 2020. Read Concern - MongoDB Manual. https://docs.mongodb.com/manual/reference/read-concern/. Accessed: 2020-05-06.

[24] MongoDB Inc. 2020. Read Preference - MongoDB Manual. https://docs.mongodb.com/manual/core/read-preference/. Accessed: 2020-05-06.

[25] MongoDB Inc. 2020. Read Preference maxStalenessSeconds. https://docs.mongodb.com/manual/core/read-preference-staleness/#replica-set-read-preference-max-staleness. Accessed: 2020-06-19.

[26] MongoDB Inc. 2020. Read Preference Use Cases. https://docs.mongodb.com/manual/core/read-preference-use-cases/. Accessed: 2020-10-13.

[27] MongoDB Inc. 2020. Replication. https://docs.mongodb.com/manual/replication/#replication-lag-and-flow-control. Accessed: 2020-05-06.

[28] MongoDB Inc. 2020. Server Selection Algorithm. https://docs.mongodb.com/manual/core/read-preference-mechanics/. Accessed: 2020-06-19.

[29] Asya Kamsky. 2019. Adapting TPC-C Benchmark to Measure Performance of Multi-Document Transactions in MongoDB. *PVLDB* 12, 12 (2019), 2254–2262.

[30] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[31] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: measuring and understanding consistency at Facebook. In *Proceedings of 25th ACM Symposium on Operating Systems Principles (SOSP'15)*. 295–310.

[32] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'18)*. 631–645.

[33] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. 2002. FAS – A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02)*. 754–765.

[34] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[35] William Schultz, Tess Avitabile, and Alyson Cabral. 2019. Tunable Consistency in MongoDB. *PVLDB* 12, 12 (2019), 2071–2081.

[36] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *Proceedings of 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, 309–324.

[37] Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, and Jack Mulrow. 2019. Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB. In *Proceedings of ACM International Conference on Management of Data, (SIGMOD'19)*. 636–650.

[38] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'17)*. 1009–1024.

[39] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'17)*. 1041.

[40] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. 2011. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective.. In *CIDR'11*, Vol. 11. 134–143.

Industrial Paper

DLC: A New Compaction Scheme for LSM-tree with High Stability and Low Latency

Peiquan Jin
University of Science and Technology of China
Hefei, China
jpq@ustc.edu.cn

Jianchuang Li
University of Science and Technology of China
Hefei, China
lijc@mail.ustc.edu.cn

Hai Long
Huawei Technologies Co., Ltd.
Shenzhen, China
longhai@huawei.com

ABSTRACT

Many big data systems employ LSM (Log-Structured Merge)-tree-based key-value stores, such as RocksDB and Cassandra. LSM-tree has a multi-level data structure and can transform random writes into sequential ones by compaction operations. However, the compaction operations in LSM-tree introduce the read/write amplification issue, which will increase the processing latency and incur throughput drops. In this paper, to eliminate the impact of compaction on the throughput stability and latency of LSM-tree, we propose a new compaction method called DLC (Delay Level-0 Compaction). We notice that workloads like OLTP are periodically high. For example, an electronic business platform may have high requests at noon and night but have few requests in the early morning. When the workload becomes high, many data will be flushed to Level-0 of LSM-tree from memory, which will trigger frequent Level-0 compaction and lower the system's throughput. The main idea of DLC is to delay Level-0 compaction at a high load and resume compaction when the system becomes low-loaded. As the low-loaded system generally has enough free CPU cores and I/O bandwidths, performing the delayed compaction will not affect the system's overall performance. Therefore, we can maintain stable throughput even when the system is high-loaded. To implement DLC, we first define a new I/O estimation model to characterize the workload. Then, we determine whether to delay Level-0 compaction according to the characteristics of the current workload. Moreover, to deal with sustained high workloads, we invent a burst compaction strategy to reduce throughput dropping and present two implementations for the bursty compaction. We implemented DLC on Myrocks and experimentally compared DLC with the original MyRocks and a state-of-the-art scheme called SILK under various OLTP workloads. The results show that DLC outperforms MyRocks and SILK in both latency and throughput stability.

1 INTRODUCTION

LSM-tree (Log-Structure Merge tree)[17] has been widely used in key-value stores, such as RocksDB and Cassandra. LSM-tree maintains a multi-level data structure, and all data in each level are stored using Sorted String Tables (SSTables), in which all key-values are sorted in order. Data are flushed from memory to the SSTables in the first level (Level-0, or L0 for simplicity) through sequential writes. As sequential writes are much faster than random writes, LSM-tree can offer high writing performance. However, when the SSTables in Level-0 exceeds a threshold, LSM-tree performs a compaction operation to merge the SSTables in Level-0 with those in Level-1. If Level-1 also exceeds the threshold, next compaction will also be triggered to merge SSTables in Level-1 with those in Level-2. During a compaction process, both CPU and the I/O bandwidth will be highly used, resulting in the decrease of the overall throughput. To validate the influence of the compaction in LSM-tree, we ran the default OLTP workload in *sysbench*[13] (see Section 6.1 for the detailed settings) on RocksDB and tested the system's throughput. As shown in Fig. 1, there are periodical throughput-drops when the system has been running for a long time, because the system has to perform periodical compaction to merge up-level data into low levels.

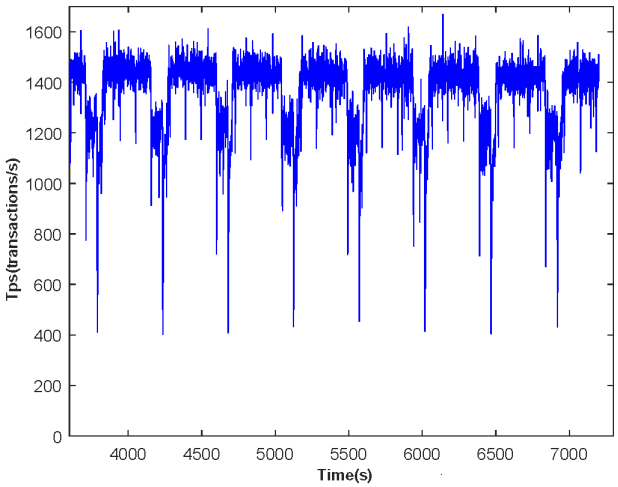


Figure 1: Throughput drops of RocksDB when running on the default OLTP workload generated by *sysbench*.

The stability of throughput is critical to many applications. For example, an online short-video social network platform like TikTok can not tolerate periodical high latency when playing videos. To improve the throughput stability of LSM-tree, various solutions [3, 6, 16, 18] have been proposed. Among them, the state-of-the-art method is SILK [3], which won the best paper of ATC 2019. The experimental results of SILK showed that it can maintain stable throughput for about 2500 seconds. However, we experimentally found that when SILK kept running for 3500 seconds, it had dramatic throughput drops and the throughput became unstable. This is mainly because the compaction scheduling in SILK can not adapt to the workload changes well.

The compaction operations in LSM-tree are known as background operations (also called internal operations), because they are scheduled on background periodically. A compaction operation consumes a great number of I/O bandwidths because it has to read and write a large amount of data. This is the main reason that affects the throughput stability of LSM-tree. The key

challenge to keep throughput stability is how to schedule compaction when the I/O bandwidth is used heavily. An intuitive solution is to reserve some I/O bandwidths for compaction operations, but different levels in LSM-tree have different needs of I/O bandwidths. Therefore, it is hard to reserve appropriate I/O bandwidths for LSM-tree to maintain throughput stability.

Basically, we can roughly divide workloads into two types, namely write-intensive workloads and read-intensive workloads. For write-intensive workloads, the competition for disk I/O between parallel compaction operations is the main reason causing the instability of throughput. In such cases, write stalls may occur and lower the throughput [16]. For this reason, most of existing studies[1, 3, 16, 18] were toward optimizing compaction for write-intensive workloads. However, so far, there is no solution that can offer continuously stable throughput for LSM-tree.

In this paper, we also focus on write-intensive workloads and aim to improve the throughput stability of LSM-tree and lower the processing latency. We propose a new compaction scheme named DLC (**D**elay **L**evel-0 **C**ompaction). DLC aims to achieve three goals. First, it should have a more stable throughput than RocksDB and SILK [3] when running for a long time. Second, it is expected to have lower latency. Third, it should adapt to periodically varying workloads, i.e., the arriving rate of requests is high for a period and then becomes low. The basic idea of DLC is to delay L0[1] compaction at a high load and to resume L0 compaction at a load load. Thus, DLC can work well on workloads with periodically varying workloads[3]. Briefly, we make the following contributions in this study:

(1) We propose a new I/O model to estimate the I/O bandwidth of the current workload effectively and precisely. Our I/O model is inspired by the model proposed by SILK [3], but we devise new estimation functions. Differing from the I/O model of SILK that simply summarizes the data size of all read and write operations, our I/O model introduces two new ideas. First, we remove the data size of write operations, because LSM-tree always writes data to in-memory Memtables, meaning that write operations will not occupy I/O bandwidths. Thus, it is not reasonable to include the written-data size in I/O bandwidth estimation. Second, we divide read operations into *Get* and *Scan* because these two read operations have different I/O costs in LSM-tree. We demonstrate that our model is more accurate than the SILK model and can allocate I/O bandwidths for background compaction more effectively.
(2) Based on the proposed I/O model, we present a new compaction scheme called DLC that delays L0 compaction at a high load and resumes L0 compaction at a load load. We use the new I/O model to characterize the I/O bandwidth need of the current workload, and determine whether the workload is high or low. When the workload is high, we delay L0 compaction. This differs from the compaction schemes in RocksDB and SILK. RocksDB uses a threshold-based compaction scheme and will trigger many compaction operations at a high load, leading to frequent throughput drops. SILK also claims to delay compaction, but it delays the bottom level[2].
(3) Although DLC works well for periodically high workloads, the SSTables in L0 may accumulate under sustained high load, leading to write stalls/stops and serious throughput drops. Thus, we devise a bursty compaction strategy to make DLC suitable for sustained high workloads. We present two implementations for the bursty compaction, namely "resume full compaction" and "resume part compaction". The former is to compact all the SSTables in L0, while the latter is to compact selected part SSTables in L0.
(4) We implemented DLC in MyRocks (MySQL with RocksDB) and evaluated DLC using the *sysbench* tool. We generate various OLTP workloads, including periodically varying workloads, workloads with different read-write ratios, workloads with a long time of a high load, and sustained high workload. We compare DLC with MyRocks and SILK (with the DLC I/O estimation model). The results in terms of throughput and latency show that DLC achieves the best throughput stability and the lowest latency in all experiments.

The remainder of the paper is structured as follows. Section 2 introduces the background and related work. Section 3 presents the I/O estimation model. Section 4 details the DLC strategy. Section 5 discusses the bursty compaction policy. Section 6 reports experimental results. And finally, in Section 7, we conclude the paper and discuss future work.

2 BACKGROUND AND RELATED WORK

2.1 LSM-tree

The basic client operations of LSM-tree are the same as the other NoSQL key-value databases[14, 23], which include *Insert, Delete, Update, Get*, and *Scan*. For convenience, we call *Insert, Delete*, and *Update* as write operations and *Get* and *Scan* as read operations. We take RocksDB as an example to introduce the LSM-tree structure. The RocksDB storage engine mainly consists of two part, Memtable and Immutable Memtables in memory and SSTables in the disk.

LSM-tree uses the Sorted String Table (SSTable) as the basic data structure, which stores key-value pairs in the disk. SSTable is a sorted table which mainly consists of data blocks and meta blocks. Meta blocks store indexes about data block and Bloom filter[5] for read. Data blocks store key-value pairs in sequence for quickly visited by read operations. SSTables are grouped by levels. We call the levels as L0, L1, ..., L_n in short from top to bottom. We call the levels near to the memory as up levels, e.g., L0, and the other levels as low levels. The SSTables in L0 are mainly flushed from Immutable Memtables in memory. The SStables in L1 and low levels are generated by major compaction.

There is one Memtable and one or more Immutable Memtables in memory. Memtable uses *Skiplist* as its structure. Skiplist is a data structure that uses probabilistic balancing. Its algorithms for insertion and deletion are much simpler and significantly faster than equivalent algorithms for balanced trees[19]. We can consider Memtable as an in-memory buffer for inserting, deleting, and updating. When Memtable reaches its capacity threshold, it will be transformed into Immutable Memtable, which cannot be modified by any (write) operations, and a new Memtable will be created for writing new key-value pairs. When a new Immutable Memtable is created, a background thread would be scheduled to flush the Immutable Memtable to disk as one SSTable, in which all the flushed key-value pairs are stored. The flush operation is also called minor compaction.

[1] L_i means Level-i in this paper.
[2] Here, Level-0 is the top level, and Level-i with the biggest i is the bottom level.

With SSTables being accumulated in the same level and reaching the threshold of the level, a major compaction will be triggered (in this paper, we simply use the term "compaction" to represent "major compaction" by default) and may schedule compaction for garbage collection to reduce disk usage and read cost[9][8]. Different levels have different thresholds for compaction, and when the SSTables in L_i or low levels reach the threshold of the level, compaction will be triggered. A compaction operation fetches one SSTable in the level, which triggers the compaction and the SSTables in the next level that have overlapping keys with the SSTable in the up level, then merges sort all key-value pairs in sequence. The merged key-value pairs are then written to new SSTables in the next level. When the flush and compaction of LSM-tree happen in the background, the system can answer write and read requests normally. The new data inserted by write operations can be put into Memtable directly, while read operations will visit Memtable, Immutable Memtables, and SSTables in all levels[9].

2.2 Compaction Optimizations for LSM-tree

2.2.1 Reducing Compaction Cost. There have been a lot of studies toward optimizing the compaction for LSM-tree. One idea is to reduce compaction cost. WiscKey[15] and HashKV[7] separate keys from values to maintain a smaller LSM-tree that contains keys and the pointer to the values, which is effective for large keys and write-intensive workloads. But their optimizations are not suitable for range scans. MonKey[9], Dostoevsky[10] change the structure of LSM-tree by tuning related parameters for different demands, and LSM-Bush[11] proposes a more general structure for more demands. Ahmad and Kemme[1] cope with spike caused by compaction by offloading compaction to a dedicated compaction server, and it solve the cache avalanche by smart warm-up strategy, which is a good method for a distributed database like HBase or Cassandra.

Some people proposed efficient scheduling algorithms to optimize compaction for LSM-tree. bLSM[20] bounds the write latency by using the spring-and-gear merge scheduler, which is not suitable for partitioning structure. Luo and Carey[16] suggest using 95% maximum throughput to run the experiments, which is adjusted to 90% in DLC experiments. dCompaction[18] proposes delayed compaction mainly for low-level compaction, which is a lazy compaction mechanism for write-intensive workloads. Chen et al.[8] uses a priority and fairness mixed compaction scheduling mechanism to reduce write amplification and read amplification.

LDC[6] decreases the tail latency and reduces write amplification by a novel Lower-level Driven Compaction, which is orthogonal to DLC. Most of them concentrate on low-level compaction and pay little attention to up-level compaction.

Other optimizations of compaction include LSbM-tree and TRIAD. LSbM-tree[21] adds one buffer for every level to decrease block cache miss and improve read performance after compaction, which concentrates on reducing cache miss after compaction. TRIAD[2] is designed for skewed workloads. It also delays L0 compaction until there is enough key overlap in L0 to be compacted. However, DLC delays L0 compaction according to the workload.

2.3 State-of-the-Art Optimization: SILK

SILK[3, 4] is the state-of-the-art optimization for the compaction in LSM-tree. In this paper, we mainly focus on improving SILK. Thus, in this section, we briefly introduce the details of SILK.

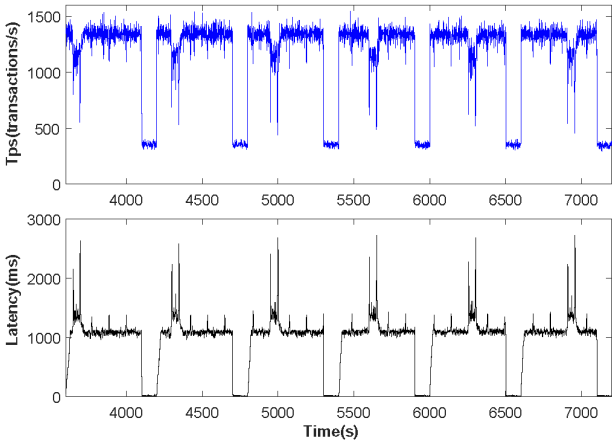


Figure 2: SILK's throughput drops and latency increasing.

SILK proposed an effective way to prevent latency spikes in RocksDB. It used an I/O scheduler for background analysis, paused scheduling low-level compaction when in high loads. The I/O scheduler in SILK can dynamically allocate bandwidth between client operations, so it can allocate more bandwidth to compaction during a low-loaded period.

2.3.1 Compaction of SILK. SILK puts forward two main methods to reduce the compaction cost in RocksDB, namely prioritizing and preempting internal operations. SILK maintains two internal thread pools, one with high priority is for flush and another with low priority is for compaction. According to the SILK I/O scheduler, when the computed bandwidth of client operations exceeds the threshold, SILK will pause compaction. As a result, only up-level compaction can be scheduled and low-level compaction will be delayed during the pausing time. Furthermore, the up-level compaction will be scheduled in the high-priority thread pools, meaning that the compaction may preempt the low-level compaction scheduled in the low-priority thread pool. The preempted compaction will recover compaction work when the thread pool is free. By this way, compaction from L0 can be scheduled along with other compaction paused, and the remaining I/O bandwidths (allocated to flush and compaction) can be allocated to the compaction from L0. When the computed bandwidth of client operations is over the threshold, SILK will resume compaction. More bandwidth will be allocated to internal operations, and parallel compaction can be scheduled. In short, SILK pauses low-level compaction at a high load and resume them at a load load.

2.3.2 Problems. As reported in the original SILK paper [3], SILK can maintain stable throughput for 2000 seconds. However, if a high workload lasts for a longer time than 2000 seconds, SILK will incur dramatic throughput drops and latency increasing. We ran SILK to see its performance and found that SILK has periodical throughput drops and latency increasing after running for more than 3500 seconds, as shown in Fig. 2. This is mainly because after a long-time running, the frequent compaction operations in SILK will consume a large amount of I/O bandwidths. In addition, the I/O analyzer of SILK fails to estimate the workload level accurately. This results in inappropriate compaction scheduling, which will finally trigger write stalls or write stops to delay writing or stop writing to the memory. For example, SILK will resume compaction even when the system runs at a high load.

Although SILK proposed to delay the low-level compaction at a high load, if the up-level compaction cannot finish in time during a load load, SSTables will accumulate in up-levels, which will incur the file retention of LSM-tree [8]. With the increasing of the number of the SSTables to be merged, SILK will finally reach the threshold of write stall or stop, resulting in throughput dropping and latency increasing.

3 I/O ESTIMATION MODEL

In this section, we propose an I/O estimation model inspired by the SILK I/O scheduler [3]. By this model, we can compute the I/O bandwidth taken by client operations more accurately than SILK. Note that the I/O bandwidth estimation is critical to compaction scheduling. If we fail to estimate the bandwidth of the current workload, we may resume compaction at a high load, like SILK, and lead to throughput drops.

3.1 The I/O Estimation Model of SILK

SILK monitors the bandwidth used by client operations and allocates the available I/O bandwidth to internal operations. It realizes its I/O scheduler by setting a separate thread on client load (which is db_bench on RocksDB actually). The I/O scheduler can get actual numbers of client operations what client has accomplished last time interval and computes client's I/O bandwidth according to Eq. 1, where N_{read} and N_{write} are the read times and write times in the last time interval, B_{kv_pairs} is the Bytes of key and value, $T_{interval}$ is the last time interval. SILK sets the limit of total bandwidth $Bandwidth_{limit}$, so it can dynamically allocate left bandwidth to internal operations easily using rate limiter according to Eq. 2, where ε is a small buffer which are not significant enough to adjust internal operation bandwidth.

$$Bandwidth_{client} = (N_{read} + N_{write}) * B_{kv\_pairs}/T_{interval} \quad (1)$$

$$Bandwidth_{internal} = Bandwidth_{limit} - Bandwidth_{client} - \varepsilon \quad (2)$$

3.2 The I/O Estimation Model of DLC

SILK has many restrictions so that it can not be applied to the OLTP workload directly. DLC optimizes the SILK I/O scheduler to make the model suit for the OLTP workload.

The separate thread used by SILK to monitor the bandwidth cannot be used for OLTP workloads.This is because SILK is toward the workloads generated by *db_bench*. But the OLTP workload is generated by *sysbench*. Therefore, we can not monitor the bandwidth of OLTP workloads with the same method as SILK. DLC uses a separate thread on RocksDB to monitor the bandwidth so that it can work on any workloads independently. Other than getting the specific counts of all client operations every time interval, which is proposed by SILK, DLC gets data from *Statistics*, which is a statistical tool provided by RocksDB. In this way, we can get the concrete counts of every client operation. As we can see in Table 1, we can get total counts of client operations according to ticker name (which is added like ticker) from *Statistics*. Based on the SILK's I/O scheduler and the total counts of client operations, we construct the I/O estimation model for DLC. This model is workload-sensitive and it can classify the current load into high or low quickly. This model has two functions, namely computation and analyzing.

3.2.1 Computation. The I/O estimation model of DLC summarizes a universal I/O cost analyzing equation based on Eq. 1 and Eq. 2. DLC computes new I/O bandwidth according to Eq. 3. In contrast to SILK, DLC adds scan s for the client bandwidth estimation because there are some scan operations on OLTP workloads. We also add weights for all client operations so as to estimate the actual I/O cost precisely. It is necessary to add weights for client operations because there exists read amplification[15]. If a *Get* operation is missed in the block cache, it will fetch at least one block from the disk. Thus, one *Get* operation will read more than one key-value pairs on average. Meanwhile, DLC ignores to compute write cost for I/O bandwidths, because write operations insert key-value pairs into memory directly, which do not consume I/O bandwidths. When flush and compaction operations happen, the bandwidth required for writing data to disk belongs to $Bandwidth_{internal}$. Though it is possible to compute the actual value of all weights for client operations, e.g., by a linear programming method, it is inconvenient in practical applications. DLC changes Eq. 3 on the basis of OLTP workloads to make it suitable for other workloads. As OLTP workloads have only one type of transaction with ten *Get* operations, four *Scan* operations, and four *Write* operations, we change Eq. 3 into Eq. 4, N_{trans} is the numbers of the committed transactions in the last time interval. Though there exists delay between operations execution and transaction commits, it is reasonable to assume that $N_{get} = 10 * N_{trans}$ and $N_{scan} = 4 * N_{trans}$. Eq. 4 shows that for OLTP with one type of transaction, all operations in the transaction have the same proportion. We only need to use one operation (*Get* or *Scan*) to compute the real client operation bandwidths. Taking the *Get* operation as an example, we can compute $Bandwidth_{client}$ easily using Eq. 5. DLC allocates the bandwidth to internal operations by using Eq. 2, which is the same as SILK.

Table 1: Some related statistical data in Statistics.

Ticker name	Description
NUMBER_KEYS_READ	Total counts of get (k)
NUMBER_KEYS_WRITTEN	Total counts of write (k, v)
NUMBER_DB_SEEK	Total counts of scan(k_1, k_2)

$$Bandwidth_{client} = (\omega_{get} * N_{get} + \omega_{scan} * N_{scan}) * \frac{B_{kv\_pairs}}{T_{interval}} \quad (3)$$

$$\begin{aligned} Bandwidth_{client} &= (\omega_{get} * 10 * N_{trans} + \omega_{scan} * 4 * N_{trans}) * \frac{B_{kv\_pairs}}{T_{interval}} \\ &= N_{scan} * (\omega_{get} * 2.5 + \omega_{scan}) * B_{kv\_pairs}/T_{interval} \\ &= N_{get} * (\omega_{get} + \omega_{scan} * 0.4) * B_{kv\_pairs}/T_{interval} \end{aligned} \quad (4)$$

$$Bandwidth_{client} = N_{get} * \omega'_{get}/T_{interval} \quad (5)$$

Though Eq. 5 is easy to compute the actual bandwidth for client operations, there is still a problem that we must compute ω'_{get} every time when we change the workload or the parameters of LSM-tree. It is difficult to change ω'_{get} when running for changeable workloads or auto-tuning LSM-tree[3]. DLC proposes a new idea to compute the actual I/O bandwidth, which is suitable for any workloads and any structures of LSM-tree. As we know, a *Get* operation gets data from the block cache and block cache gets blocks from the disk when a cache miss occurs, so the actual I/O bandwidth is consumed when the block cache gets blocks from the disk. DLC computes the actual I/O bandwidth by

[3]Changeable workload means the proportion of operations can change, auto-tuning LSM-tree means that the parameters of LSM-tree can be changed when running.

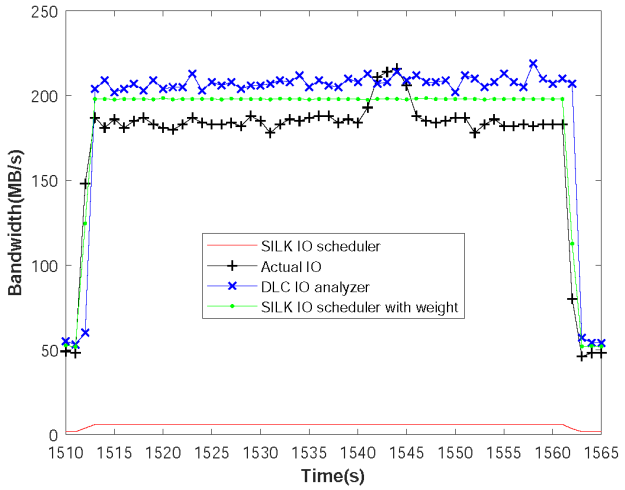


Figure 3: Comparison of various I/O estimation models (using *iostat* on RocksDB).

counting the number of the blocks added to the block cache in the time interval, as we can see from Eq. 6, where γ is the compression ratio of SSTable[4], $Sum_{block\_cache\_added}$ is the number of the blocks added to the block cache, and B_{block} is the bytes within one block.

$$Bandwidth_{client} = \gamma * Sum_{block\_cache\_added} * B_{block}/T_{interval} \quad (6)$$

We conducted an experiment on RocksDB to compare different I/O estimation models with the actual I/O bandwidth. The results are shown in Fig. 3. We can see that the SILK I/O estimation model can not suit for the workload while the DLC I/O scheduler and the SILK I/O scheduler with weights can estimate the actual I/O bandwidth with a little tolerable error, both of which can be applied to our experiments. The DLC I/O scheduler is easier to use than the SILK I/O scheduler with weights.

3.2.2 Analyzing. SILK uses a simple threshold of bandwidth to distinguish between a high load and a load load. It computes $Bandwidth_{client}$ every 10ms and allocates the remaining bandwidth to flush and compaction. DLC adds some parameters for tuning the model, which we can see in Table 2. DLC uses these parameters to distinguish a high load and a load load . For every time_interval, DLC computes the bandwidth of client operations according to Eq. 5. If the bandwidth exceeds the io_high_limit, DLC will regard the workload as a high load. However, low bandwidth is not a sufficient condition of a load load because there are many reasons like flush and compaction that will lead to low bandwidth over a period of time. Thus, even if the client bandwidth is inferior to io_low_limit, we can not judge that it is a low load. DLC uses *softness* to determine the sensitivity of the model. Only when we get several continuous times of low bandwidth can we conclude that the current workload is a low load. By using these parameters, DLC can classify the current workload more correctly than SILK.

[4]Because data in SSTable is compressed while data in the block cache is uncompressed.

4 DESIGN OF DLC

DLC is an improved version of SILK (also an optimization of RocksDB), which optimizes the I/O scheduler of SILK and proposes to delay L0 compaction, which differs from SILK that delays low-level compaction.

4.1 DLC's Compaction Policy

DLC proposes a novel idea to reduce the compaction impact to the throughput stability. The main idea of DLC is to delay L0 compaction[5] at a high load, and to resume L0 compaction when workload is low. DLC uses the I/O estimation model discussed in Section 3.2 to compute the I/O bandwidth, analyze the workload, and decide whether to delay or resume L0 compaction. As we described before, the DLC I/O estimation model can judge the workload more correctly than SILK. First, in the case of fluctuation of workload, DLC sets the time_interval to 1s by default, which can fit most workloads (including OLTP) because many of them also conclude and summarize the statistical data every one second by default. The other parameters in DLC are set according to the workload so that DLC can distinguish the actual high or low workload correctly. Second, on the basis of the actual workload, DLC gives two optional policies for scheduling compaction in a sustained high load, namely delay full L0 compaction or delay part L0 compaction.

To delay full L0 compaction means to delay compaction from L0 to both L0 and L1 under a high load and to resume compaction under a load load. Other than changing the threshold of L0, DLC changes the scheduling mechanism so that it can really delay compaction. Because of the compaction mechanism of RocksDB, up-level compaction will be triggered unexpectedly. Increasing the threshold of L0 can only delay compaction temporarily but the compaction will still be triggered, which is almost uncontrollable. DLC proposes a controllable delay mechanism so that it can delay compaction at a high load and resume compaction at a load load.

To delay part L0 compaction means DLC only delays compaction from L0 to L1. It is a trade-off between read performance and throughput stability. With the accumulation of the SSTables in L0, the read latency will increase and the throughput will decrease. Thus, DLC allows compaction to be scheduled from L0 to L0, which will impact the temporary throughput in a short time but is helpful for future performance. To delay part L0 compaction is suitable for a high load with a relatively long time, which also needs a longer low load to resume compaction. However, this method can not stop L0 compaction fully, so the throughput may drop when compaction from L0 to L0 happens. Therefore, we take "delay full L0 compaction" as the default policy. For both policies, we allow only one low-level compaction under a high load being scheduled.

DLC assumes that there are only a few bandwidths that can be allocated to internal operations at a high load and the flush operations from Immutable Memtable to L0 is unstoppable. Thus, there must be some bandwidths allocated to flush, which may cause a small fluctuation when flush happens. Compared to flush operations and low-level compaction, up-level compaction will cause longer time and bigger fluctuation. The reasons are as follows. First, up-level compaction will merge more SSTables than flush. Second, the key range of up-level compaction is wider than low-level compaction, and the access frequency of the up-level SSTtables is higher than the low-level files[9].

[5]We also call L0 compaction as up-level compaction in this paper.

Table 2: Some parameters added by DLC.

Parameter name	Description	Default Value
io_low_limit	The limitation of io bandwidth to confirm the low load	180MB/s
io_high_limit	The limitation of io bandwidth to confirm the high load	300MB/s
io_limit	The io bandwidth of disk	350MB/s
softness	The sensitivity of DLC to confirm low or high low	3
time_interval	The time interval to compute and allocate the bandwidth	1000ms
ω'_{get}	The modified weight of get operation	16000

The throughput will decrease if up-level compaction is scheduled at high load. Because up-level compaction will occupy a large quantity of I/O bandwidths, client operations cannot get enough bandwidths, yielding the drops of throughput. Thus, DLC delays up-level compaction until the next low load is detected. When there are plenty of bandwidths allocated to compaction at a load load, DLC resumes compaction to make full use of the bandwidth.

4.2 Rate Limiter for DLC

When flush happens, it merge-sorts key-value pairs in all Immutable Memtables (if only one Immutable Memtable, no merge-sort will happen) and writes them to a new SSTable in L0, so flush consumes disk I/O write bandwidth only. But when compaction happens, it first reads related SSTable from disk, merge-sorts them and writes them to a few SSTables in the corresponding level, so compaction consumes both disk I/O write and read bandwidth. Rate limiter is designed to throttle the maximum write speed within a certain limit for lots of reasons. For example, flash writes cause terrible spikes in read latency if they exceed a certain threshold. In other words, the rate limiter is to limit the speed of the data written to the disk. And the rate limiter can be modified for throttling the maximum sum speed of both read and write easily, which is used in DLC. By dynamically allocating left bandwidth is available for flush and low-level compaction but cannot quite effective for up-level compaction because bandwidth is not the only reason for the fluctuation of throughput[1, 21]. The bandwidth exceeds the limit of speed to cause terrible spikes, leads to terrible fluctuation and long latency. So to delay L0 compaction under high load may be the best for OLTP workload to maintain both high throughput and low latency and to resume L0 compaction under a load load to achieve minimal losses of throughput and latency.

5 BURSTY COMPACTION FOR DLC

DLC is mainly designed for the OLTP workload with periodical high and low loads. By delaying L0 compaction at high load and resuming L0 compaction at a load load, we can make full use of the I/O bandwidth with the least throughput loss. However, if the workload becomes continuously high, which is called a sustained high load in this paper, the SSTables in L0 will become more and more, leading to write stalls or stops. When running under a sustained high load, DLC will keep delaying L0 compaction to maintain throughput. The read performance will become worse and the throughput will decrease gradually as time goes. There is no time and bandwidth for compaction at a sustained high load; all read operations amortize the influence of delaying L0 compaction. This problem could be solved when the workload changes into a load load and DLC resumes L0 compaction. However, when running with a sustained high load, the workload will not change into a load load, which will gradually affect throughput of DLC, and cause some other unavoidable result such as write stall or write stop[8].

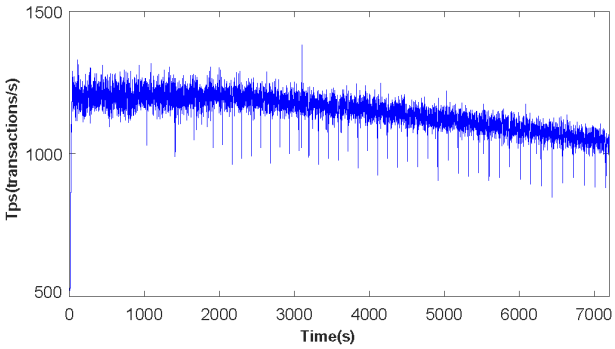


Figure 4: DLC running on the sustained high workload.

Generally, the system will not always be at a high load. Thus, we can assume that the system's throughput is limited by the number of the accumulated SSTables in L0. However, if the workload keeps high for a long time, we can infer that the system's throughput will finally decrease because more and more SSTables will be accumulated in L0. To verify our analysis, we tested DLC under a sustained high workload and the result is shown in Fig. 4, which shows that the throughput decreases with time. If we do not take any action, the sustained high workload will finally trigger write stalls or stops, which will worsen the performance of DLC.

To make DLC suitable for sustained high workload, we further propose a bursty compaction policy, which can avoid the throughput drops of DLC under the sustained high workload. The idea of bursty compaction is to compact selected SSTables in L0 to avoid the continuous accumulation of SSTables and triggering write stalls or stops. This is implemented by monitoring a threshold representing the number of accumulated SSTables in L0.

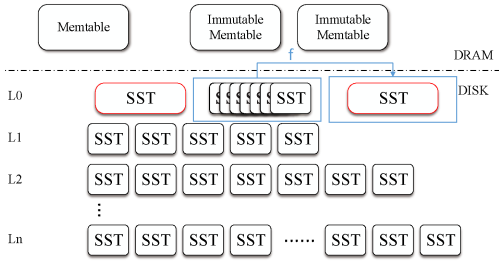


Figure 5: Bursty compaction from Immutable Memtable to L0.

Resume Full Compaction. To resume full L0 compaction is that when the amount of SSTables in L0 gets the threshold, or the total size of SSTables in L0 gets the threshold, DLC stops delaying and resumes compaction from L0 to L1, this cumulatively bursty compaction would consume plenty of time and I/O bandwidth, causing an inevitable degradation of throughput and increase in latency. As we can see in Fig. 4, we could not maintain high throughput all the time and we should schedule compaction in time to keep high throughput and low latency by sacrificing performance for a period of time.

Resume Part Compaction. To resume part L0 compaction is that when the number of the SSTables in L0 flushed from MemTable gets the threshold, DLC resumes compaction from Immutable Memtable to L0. The difference between resume part L0 compaction in the bursty compaction and normal compaction from Immutable Memtable to L0 in MyRocks is that our bursty compaction only merges and sorts the SSTables flushed from Immutable MemTable but normal compaction will merge and sort all SSTables in L0. As compaction will produce a big SSTable that reserves in L0, the bursty compaction does not merge all SSTables to reduce disk I/O bandwidth. Figure 5 shows the idea of the bursty compaction. The SSTable generated from the bursty compaction will not be scheduled again in future bursty compaction.

By controlling the parameters of the threshold, DLC can schedule both two policies easily or only one policy according to the need. Bursty compaction is a new compaction mechanism for sustained high workload, which permits schedule up-level compaction temporarily for future throughput and latency with an inevitable fluctuation for a period of time.

6 PERFORMANCE EVALUATION

6.1 Experimental Setting

We conducted experiments on the Elastic Cloud Server of Huawei Cloud running Linux CentOS 7.6. The server has four Intel Xeon 4-core CPUs with 3.0GHz and 32GB of DRAM. It has one 128GB super-high SSD for storing logs and another 640GB super-high SSD (350MB/s approximately) for storing the MyRocks data.

We use 64 tables in the experiment and each table has 10^7 records. Each key-value pair has a 16B key and a 184B value. We set a 128MB MemTable and only one Immutable Memtable. We set the threshold of L1 to 2GB, the size of SSTable to 64MB, the size-ratio of each adjacent levels to 10, the level of LSM-tree to 5, the block-cache size to 12GB, and the block size to 64KB. The database size is nearly 140GB. We set level0_slowdown_writes_trigger and level0_stop_writes_trigger to 100 both to avoid write stalls or write stops too early.

We compare DLC with two competitors, including MyRocks (MySQL 5.7.26-29 with RocksDB) and SILK. To make the comparison fair, we replace the I/O estimation model of SILK with the DLC I/O estimation model, and we use SILK*(SILK with the DLC I/O estimation model) to indicate this modification.

We use OLTP in *sysbench* [13] as the basic benchmark. The OLTP (Online Transaction Processing) workload in *sysbench* [13] is a SQL workload that can be adjusted to read-intensive or write-intensive. The OLTP workload in *sysbench* has only one type of transaction, which has ten point queries, four range queries, two update queries, one delete query, and one insert query. By using *sysbench*, we can generate OLTP workloads with periodic high and a load loads, which can satisfy most experiments in this paper.

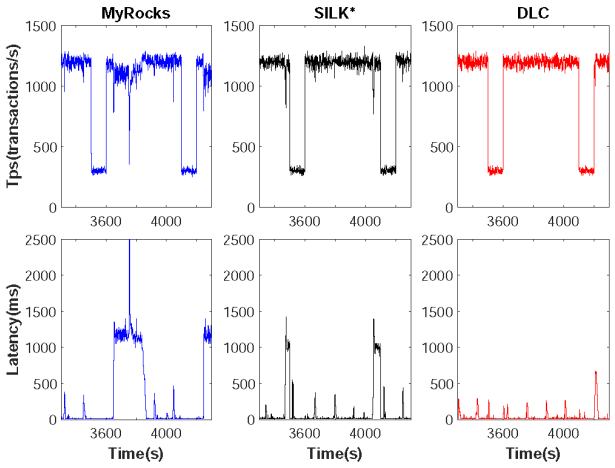


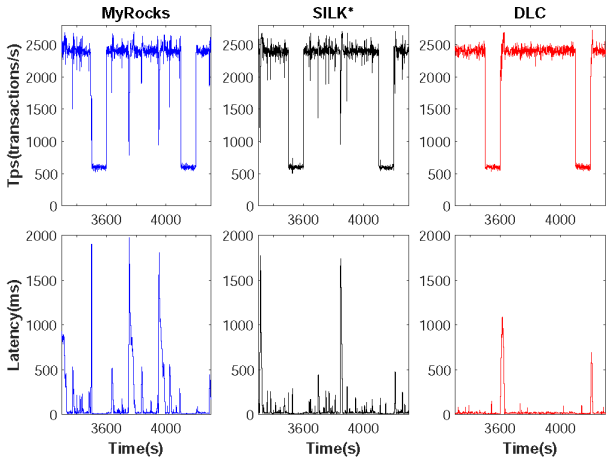
Figure 6: MyRocks, SILK* and RocksDB running on the default OLTP workload.

To make the workload not overwhelm the maximum capacity of the system, in our experiments, we first run the workload to measure the maximum throughput of the system, then we set 90% maximum throughput as the threshold to ensure that the workload will not overwhelm the capability of the system. We mainly evaluate two metrics, namely throughput and latency. We use transactions per second to represent throughput, and use the P99 latency to represent latency. The P99 latency refers to the 99th latency percentile, meaning that 99% of requests(transactions) will be faster than the given latency number, and only 1% of the requests will be slower than the P99 latency. These metrics have also been used in prior work like SILK[3][4].

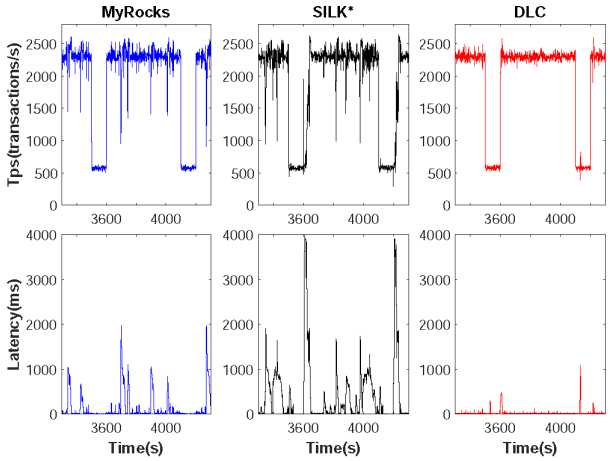
6.2 OLTP with Periodically Varying Workloads

Both DLC and SILK are designed toward periodically varying workloads, i.e., the arriving rate of requests is high for a period and then becomes low. Note that a continuously-high workload will overwhelm the maximum capacity of the system. In this situation, all approaches will fail to keep a stable throughput. On the other hand, most big-data applications like E-commerce platforms have the feature of periodically varying workloads. Another assumption of DLC and SILK is that the workload is write-intensive. This is because only frequent writes can trigger frequent compaction operations, which can be utilized to evaluate the performance of DLC and SILK. How to avoid throughput drops caused by compaction is more challenging that other issues in current LSM-tree-based systems. Although it is important to optimize the read performance of LSM-tree, e.g., under read-intensive workloads, it is orthogonal to this study. An intuitive way to improve read performance is to enlarge the block cache.

To generate appropriate workloads for DLC and SILK, we run the default OLTP workload (each transaction has ten point queries, four range queries, two update queries, one delete query, and one insert query.) in *sysbench* with a high arriving rate for 500s, followed by a low arriving rate for 100s. The high arriving rate is set to 1,200 transaction per second, and the low arriving rate is 300 transactions per second. Note that the two rates and the time period for high/low should be set according to the maximum capacity of the system to be evaluated. In our experiment, the time interval of two up-level compactions is between 400s and



(a) R:W = 5:5



(b) R:W = 1:9

Figure 7: MyRocks, SILK*, and DLC under different read-write ratios.

600s. We list the parameters of the DLC's I/O analyze model on Table 2. We execute the workload on MyRocks, SILK*, and DLC and calculate the throughput and latency continuously. Since at the beginning there is no compaction triggered, we only report the results of all systems between 3,300s and 4,300s. When each system has been running for over 3,000s, we notice that there will be frequent compaction triggered by the insertions of key-value pairs.

6.2.1 Under the Default OLTP Workload. Figure 6 shows the throughput and latency of MyRocks, SILK*, and DLC under the default OLTP workload. In this experiment, the threshold of the maximum capacity of the system is set to 1,200 tps. When the throughput is high (about 1,200 tps in the figure), the system runs at a high load. When the throughput is low (about 300 tps), the system runs at a load load. This is consistent with the periodically varying feature of the workload. MyRocks shows the worst performance. It can not keep stable throughput at a high load because it has to perform up-level compaction at a high load, which will consume additional system resources (I/O bandwidth, CPU, and memory) and lower the throughput. This also leads to the high latency of MyRocks. The throughput stability and latency of SILK* is better than MyRocks, owing to the delay of low-level compaction in SILK*. However, SILK* has a lot of throughput drops during the high-load period. We can see in Fig. 6 that there are serious throughput drops near 3,500s and 4,100s. This is mainly because when the high load runs for a long time, many up-level compactions have been triggered, but SILK* has to perform those up-level compactions even when the system runs at a high load, which also leads to the increasing of the latency of SILK*. On the contrary, DLC exhibits the most stable throughput and the lowest latency compared to MyRocks and SILK*. When the system runs at a high load, DLC can always keep the throughput around 1,200 tps, which is the arriving rate of the high load. When the system runs at a load load, DLC can keep the throughput around 300 tps, which is the arriving rate of the low load. We can see in the figure that DLC has no serious throughput drop. Moreover, the latency of DLC is much lower than others, because it always performs up-level compaction at a load load. In summary, DLC achieves a more stable throughput and higher time performance than its competitors.

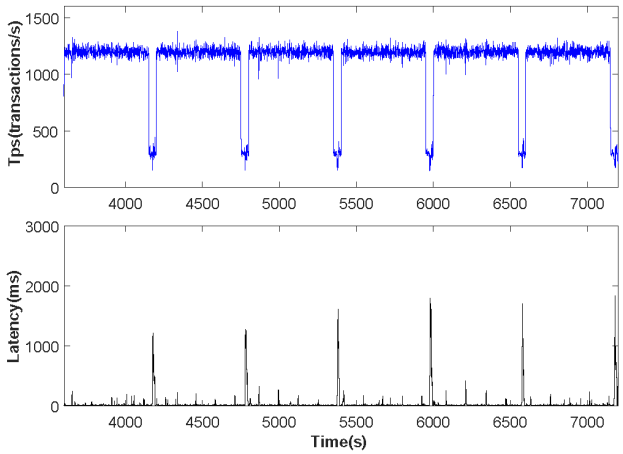
6.2.2 Varying the Read-Write Ratio. In this experiment, we test the performance of DLC under OLTP workloads with different ratios of read and write requests. As DLC is proposed for write-intensive workloads, we prepare two types of OLTP workloads with a read-write ratio of 5:5 and 1:9. Note that in this experiment, we remove the range and update queries from the OLTP workloads to make the workload easier to be generated. When the read-write ratio is set to 5:5, the threshold of the maximum capacity is set to 2,500 tps, which is determined by running the workload before the experiment. When the read-write ratio is 1:9, the threshold is set to 2,350 tps. Figure 7 shows the throughput and latency of MyRocks, SILK*, and DLC under the two read-write ratios. We can see that DLC performs better under the 1:9 read-write ratio, showing that DLC is more efficient for write-intensive workloads. For the workload with the 5:5 read-write ratio, DLC also achieves the best stable throughput and the lowest latency than MyRocks and SILK*.

6.2.3 High Load with a Long Time. Next, we evaluate the performance of DLC under a long period of a high load. This experiment is to show whether DLC can still keep high performance under a long time of a high load. For this sake, we also use the default OLTP workload but shorten the time period of a load load to only 50s, which means that we leave little time for DLC to perform delayed up-level compaction. In addition, we change the time period of a high load to 550s and 1,100, respectively. Consequently, we get two workloads, one is with 550s high load followed by 50s low load, and the other is 1,100s high load followed by 50s low load.

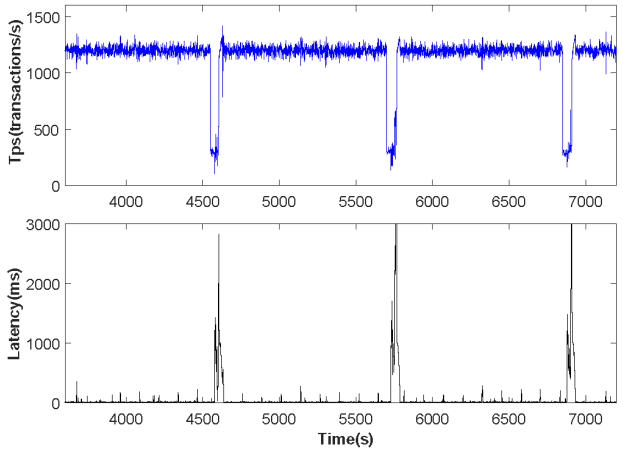
Figure 8 shows the throughput and latency of DLC under the two kinds of workloads. We can see that DLC maintains a stable throughput even when the high-load period increases from 550s to 1,100s, indicating that DLC can adapt to workloads with varying periods of a high load. This also shows that the compaction scheduling cost of DLC is relatively low and DLC can quickly detect the status change of the workload and perform the delayed up-level compaction. Note that DLC has periodic high latency arising under the 1,100s high load, as shown in Fig. 8(b). This is because there are more accumulated SSTables in L0, which cost more time of DLC to complete the compaction.

6.3 Performance of Bursty Compaction

In this experiment, we verify the efficiency of the bursty compaction of DLC. When the workload becomes continuously high

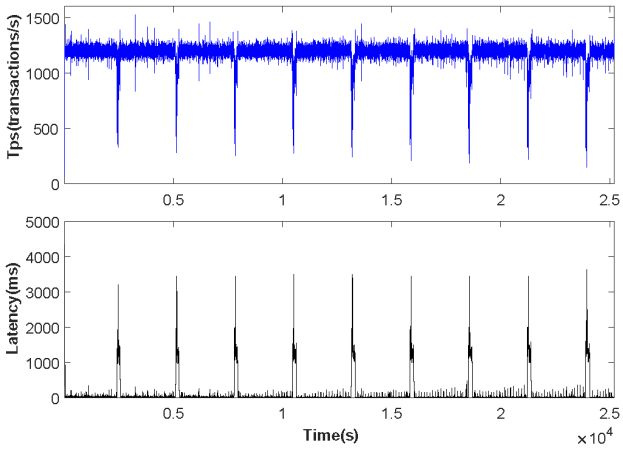


(a) 550s high load

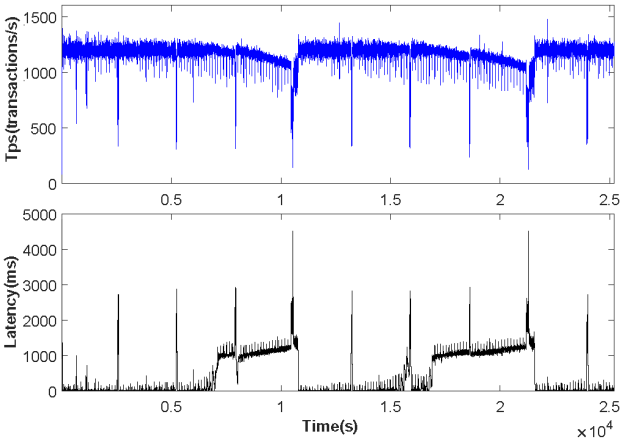


(b) 1100s high load

Figure 8: DLC under high load with a long time.



(a) resume full compaction only



(b) resume part and full compaction

Figure 9: Performance of DLC on a sustained high load.

(which is named sustained high load), the accumulated SSTables in L0 will become more and more, which will worsen the read performance and lower the throughput. DLC monitors the number of the accumulated SSTables in L0, and if the number exceeds a threshold, DLC will perform the bursty compaction to merge selected SSTables in L0 to L1.

To generate a sustained high load, we run the default OLTP workload continuously at the high arriving rate (1,200 tps), and let the system run for a long time to make the number of the SSTables in L0 increase to the threshold (which is set to 20 in this experiment). Figure 9 shows the throughout and latency trend of DLC under a sustained high load. Figure 9(a) shows the result of the "resume full compaction" policy, which is to resume full L0 compaction when the number of SSTables flushed from MemTable exceeds the threshold. Figure 9(b) shows the result of the "resume part and full compaction" policy, which is to resume part L0 compaction when the number of SSTables flushed from MemTable reaches the threshold and to resume full L0 compaction when part compaction has been scheduled for four times.

Figure 9 shows that both the two policies can maintain stable throughput for about 2,550s with a short time of throughput degradation (about 60s for the "resume full compaction" and about 40s for the "resume part compaction"). The "resume full compaction" policy can quickly resume high throughput after bursty compaction, but it has to compact all SSTables in L0, which is time-consuming. We can see in Fig. 9(a) that the latency of the "resume full compaction" becomes extremely high when DLC performs the full compaction. On the other hand, the "resume part compaction" policy only compact selected partial SSTables for maintaining a stable throughput of DLC. Thus, the cost of part compaction is lower than that of full compaction. As shown in the figure, the latency of part compaction is lower than that of full compaction. However, the "resume part compaction" policy sacrifices part of the read performance (read requests still need to read many SSTables in L0), resulting slightly dropping of throughput. To avoid continuous throughput-drops caused by the accumulation of the SSTables in L0 (even after part compaction), the "resume part compaction" in DLC performs full compaction when part compaction has been scheduled for four times. As shown in Fig. 9(b), the throughput slightly drops with time but resume to a high level after four part compactions (each serious drop in the figure indicates part compaction).

6.4 Impact on Read Performance

In this experiment, we measure the impact of DLC on the read performance. Basically, as DLC delays the up-level compaction, there may be accumulated SSTables in L0, which will worsen the read performance under read-intensive workloads.

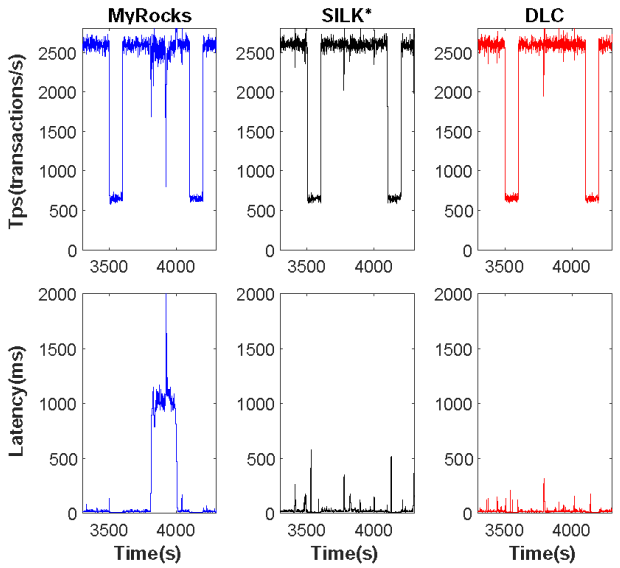


Figure 10: MyRocks, SILK*, and DLC under read-intensive workload with 90% reads and 10% writes.

We first modify the OLTP workload used in Fig. 7 by changing the read-write ratio to 9:1, preparing a read-intensive workload. Then, we run MyRocks, SILK*, and DLC to compare the throughput and latency. The results are shown in Fig. 10. Although DLC shows worse performance compared to its performance under write-intensive workloads (see Fig. 7), it still has comparable throughput stability with SILK*, and its latency is lower than that of SILK* and MyRocks. Thus, DLC can also work for read-intensive workloads.

Further, to measure the number of the SSTables in L0, we conduct an additional experiment to see the change of the number of L0 SSTables in DLC. In this experiment, we use the default benchmark tool *db_bench* in RocksDB and simply run DLC on RocksDB to calculate the number of the SSTables in L0 while DLC is running. We set one thread for inserting key-value pairs and thirty threads to perform *Get* operations. Figure 11 shows the change of the number of the SSTables in L0 as well as the read performance (in terms of QPS, because *db_bench* does not support multi-transaction processing). We can see that the number of L0 SSTables increases with time stably. Note such increase is not a linear function. We explicitly show a part of the enlarged curve in the figure, indicating that the increasing of SSTables is step-wise. This is because only when we flush Immutable Memtable to L0, the number of SSTables in L0 can increase. With the accumulation of the SSTables in L0, QPS slightly decreases while the read latency increases. Figure 12 shows the change of the number of the SSTables in L0 as well as the read performance when we use *Scan* operations and other settings remain unchanged, which shows similar results as Fig. 11.

In summary, DLC is especially suitable for write-intensive workloads, but it can also maintain comparable performance with SILK* under read-intensive workloads. Although the delay of up-level compaction results in the accumulation of the SSTables in L0, DLC can merge them to L1 at a load load or by performing bursty compaction.

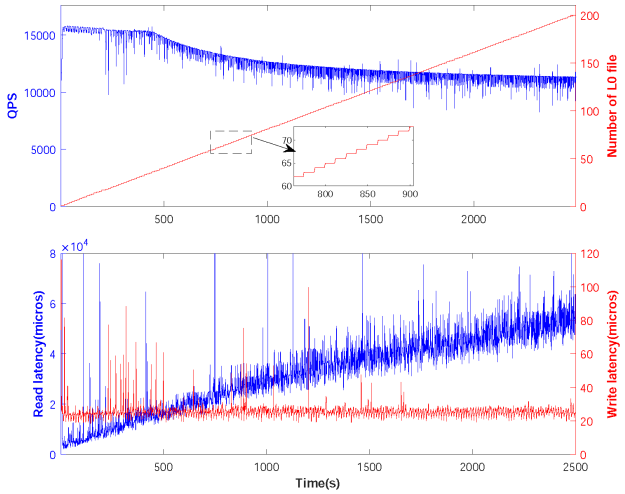


Figure 11: Accumulation of L0 SSTables and its impact on *Get* performance.

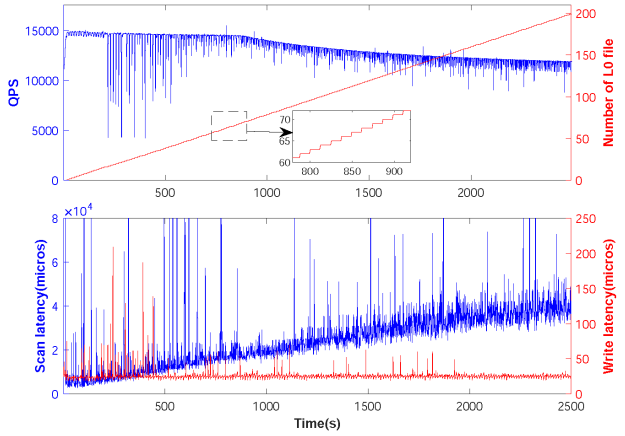


Figure 12: Accumulation of L0 SSTables and its impact on *Scan* performance.

7 CONCLUSION AND FUTURE WORK

LSM-tree has been widely used in many key-value stores, due to its high writing performance. However, the compaction operations in LSM-tree highly impact the throughput of LSM-tree, especially when LSM-tree runs under write-intensive workloads. Prior work has shown that compaction will result in serious throughput drops and increasing in processing latency. In this paper, aiming to provide stable high throughput and low latency, we proposed to delay the L0 compaction in LSM-tree when the system is at a high load and perform the delayed L0 compaction at a load load. With such a mechanism, the system's throughput can maintain a high level at a high load because no up-level compaction will be executed. On the other hand, performing compaction at a load load has little impact on the throughput because the system' resources, including I/O bandwidth and CPU, are not fully used.

Following the idea of delaying L0 compaction, we presented the DLC approach to optimize the compaction scheme in LSM-tree. We first proposed a new model to estimate the I/O bandwidth that is needed by the workload. Based on the I/O estimation model, DLC decided whether to delay the up-level compaction or

to perform the delayed compaction. DLC is especially designed for periodically varying workloads, i.e., the arriving rate of requests is high for a period and then becomes low. By scheduling up-level compaction appropriately, DLC can main stable throughput and latency. Further, to solve the problem that the workload is continuously high for a long time, which is called a sustained high load in the paper, we proposed the bursty compaction policy to perform mandatory compaction of the SSTables in L0, so as to avoid the drops of the throughput. We designed two policies to implement the bursty compaction, namely "resume full compaction" and "resume part compaction". The difference between the two policies lies in the range of the L0 SSTables to be compacted.

Finally, we implemented DLC on RocksDB and compared DLC with MyRocks and SILK* (SILK with the DLC I/O estimation model), which is the state-of-the-art optimization of the compaction in LSM-tree. The experimental results under different kinds of OLTP workloads suggest that DLC has the best throughput stability and the lowest latency. We also demonstrated that DLC can achieve comparable performance with SILK* under read-intensive workloads.

In the future, we will consider optimizing the read performance of LSM-tree and building a read/write-optimized tree structure[12]. The current design of DLC is not read-friendly, making it more suitable for write-intensive workloads. We will focus on improving the block cache management scheme[22] and the Bloom filter to reduce the read amplification and block-cache miss in LSM-tree.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable comments on this paper. We are also grateful to Prof. Jianliang Xu for his constructive suggestions on improving the paper. This work is partially supported by the National Science Foundation of China (No. 62072419 and No. 61672479) and Huawei Technologies Co., Ltd.

REFERENCES

[1] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment* 8, 8 (2015), 850–861.

[2] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of 2017 USENIX Annual Technical Conference (ATC)*. 363–375.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of 2019 USENIX Annual Technical Conference (ATC)*. 753–766.

[4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, and Diego Didona. 2020. SILK+ Preventing Latency Spikes in Log-Structured Merge Key-Value Stores Running Heterogeneous Workloads. *ACM Transactions on Computer Systems* 36, 4 (2020), 1–27.

[5] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[6] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, Ning Bao, and Yushi Liang. 2019. LDC: A Lower-Level Driven Compaction Method to Optimize SSD-Oriented Key-Value Stores. In *Proceedings of the 35th International Conference on Data Engineering (ICDE)*. IEEE, 722–733.

[7] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of 2018 USENIX Annual Technical Conference (ATC)*. 1007–1019.

[8] Lidong Chen, Yinliang Yue, Haobo Wang, and Jianhua Wu. 2018. A Priority and Fairness Mixed Compaction Scheduling Mechanism for LSM-tree Based KV-Stores. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*. Springer, 89–105.

[9] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*. ACM, 79–94.

[10] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. ACM, 505–520.

[11] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. 449–466.

[12] Peiquan Jin, Chengcheng Yang, Christian S. Jensen, Puyuan Yang, and Lihua Yue. 2016. Read/write-optimized tree indexing for solid-state drives. *The VLDB Journal* 25, 5 (2016), 695–717.

[13] Alexey Kopytov. [n.d.]. Sysbench. https://github.com/akopytov/sysbench.

[14] Ruicheng Liu, Peiquan Jin, Xiaoliang Wang, Zhou Zhang, Shouhong Wan, and Bei Hua. 2019. NVLevel: A high performance key-value store for non-volatile memory. In *Proceedings of the 21st IEEE International Conference on High Performance Computing and Communications (HPCC)*. IEEE, 1020–1027.

[15] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage* 13, 1 (2017), 5.

[16] Chen Luo and Michael J. Carey. 2019. On Performance Stability in LSM-based Storage Systems. *Proceedings of the VLDB Endowment* 13, 4 (2019), 449–462.

[17] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[18] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed compaction for the LSM-tree. *International Journal of Parallel Programming* 45, 6 (2017), 1310–1325.

[19] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990).

[20] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 International Conference on Management of Data (SIGMOD)*. ACM, 217–228.

[21] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *Proceedings of the 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 68–79.

[22] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proceedings of the VLDB Endowment* 13, 11 (2020), 1976–1989.

[23] Zhou Zhang, Peiquan Jin, Xingjun Hao, Ruicheng Liu, Xiaoliang Wang, and Shouhong Wan. 2019. RadixKV: A memory efficient and high performance key-value store. In *Proceedings of the 21st IEEE International Conference on High Performance Computing and Communications (HPCC)*. IEEE, 2774–2781.

Industrial Paper

Financial Data Exchange with Statistical Confidentiality: A Reasoning-based Approach*

Luigi Bellomarini
Banca d'Italia

Livia Blasi
Banca d'Italia

Rosario Laurendi
Banca d'Italia

Emanuel Sallinger
TU Wien and University of Oxford

ABSTRACT

Confidentiality is a crucial requirement in financial data exchange processes. On the one hand, rich microdata is needed for most AI applications, including banking supervision, anti-money laundering, etc. On the other hand, organizations may not be legally authorized to see particular data, e.g., personal data. Striking the right balance provides a number of challenges.

Motivated by our experience with the Central Bank of Italy, in this work we present Vada-SA, a reasoning-based framework for financial data exchange with statistical confidentiality. We present a production-ready and fully engineered framework, adopting a reasoning approach. The framework includes explicit consideration of the reasoning process, the business context and declarative transparency that puts the user in control. We show and discuss a number of risk measures and anonymization criteria, implemented and operated in practice.

1 INTRODUCTION

Confidentiality in financial data exchange has multiple facets and touches different business segments of the FinTech area. In *open banking* settings, where the increasingly frequent interactions between financial intermediaries motivated by the unbundling and rebundling of the banking process sees the interplay of many actors, each interested in utilizing the data about a specific portion of the process, but with limited or no access-rights to the identity of the involved customers; in European-level *banking supervision*, where data exchange between the European Central Bank and the National Central Banks needs to reveal situations that are highly critical in terms of the "financial health" of the banks, while the identity of the involved customers tends to be irrelevant; in *anti-money laundering*, where most modern approaches pinpoint fraudulent or collusive cases by inspecting high-level features of the considered actors, without accessing their identity before any judicial or law-enforcement action authorizes it; in *statistical and economic research*, with the more and more common establishment of national "Research Data Centers", data archives used by financial authorities that wish to share relevant financial data with universities and research institutes while keeping personal data reserved. Moreover, it goes without saying that the *GDPR regulation* makes the attention to the confidential transfer of personal data a central topic in Europe.

As a matter of fact, financial and statistical authorities and intermediaries look at solutions to share their own *microdata*, i.e., non-aggregated data at the finest level of granularity, while striking a good balance between their statistical relevance and the need to eliminate any possible trace of personal identities. Many situations arise in the financial segment in which a counterparty must at the same time see parts of the data (to carry out a portion of the process) and must not see other parts which they are not legally authorized to see, e.g., personal data.

This paper is motivated by our experience with the Central Bank of Italy, which, in its capacity of national central bank, banking supervision and oversight authority and Financial Intelligence Unit for Italy, is touched by the problem of *confidential financial data exchange* in all its perspectives. In this work, we present Vada-SA, the joint effort of the Applied Research Team of the Bank of Italy, TU Wien and the University of Oxford towards a reasoning-based approach to the problem.

The desiderata. We start by laying out the main desiderata for a state-of-the-art financial data exchange solution with confidentiality: (i) It should be *context aware* and take into consideration the specific business domain and the characteristics of the involved entities and features to evaluate the risk of a breach of confidentiality; (ii) At the same time it should be *schema independent*, and operate regardless of the specific dataset structure; (iii) It should be *preemptive*, in the sense that it should be able to analyze a given dataset to be exchanged and provide a confidentiality score beforehand, so that analysts can evaluate the risk of sharing it; (iv) It should be *active*, in the sense that whenever the confidentiality score is over a certain threshold (e.g., statistically inferred or defined by the domain experts), the solution should be able to alter the data and *anonymize* them so that the threshold is respected; (v) It should embody a *statistics-preserving* anonymization logic, by removing the minimum amount of information needed to guarantee confidentiality, while preserving the statistical soundness and relevance of data; (vi) It should be *fully explainable*, meaning that the confidentiality score of a candidate dataset as well as the reasons for specific anonymization choices should be completely understandable to domain experts; plus it should have a transparent semantics of confidentiality; (vii) It should be *business friendly*, by being extensible, IT-independent and at business level, i.e., domain experts should operate autonomously in defining new scoring criteria as well as anonymization logic in a high-level non-technical language; (viii) It should be *scalable* and able to handle increasing data volumes.

Statistical Disclosure Control. The area of *Statistical Disclosure Control* [26, 35, 37] (SDC) represents a relevant yardstick for our work. The SDC approach concentrates on *re-identification*, i.e., the possibility for an attacker to cross-link information it rightfully retains, in particular, every single tuple of a legitimately owned database, with other data sources so as to find out the underlying identities (of the involved people, companies and stakeholders in general). SDC adopts quantitative indicators to take decisions on data sharing by evaluating the *risk of re-identification* and balancing it with the measure of the statistical

*The views and opinions expressed in this paper are those of the authors and do not necessarily reflect the official policy or position of Banca d'Italia.

relevance of the data, so as to minimize the risk while maximizing the *statistical utility*. SDC also studies solutions to transform, namely *anonymize*, the data to be shared, balancing confidentiality and statistical relevance. Commonly adopted techniques, featured by widespread tools such as *sdcMicro* [9], *μ-ARGUS* [27], and *ARX* [33], aim at removing potential identifiers (sometimes known as *quasi-identifiers*) of the disclosed tuples and include *value suppression*, *aggregation*, and *generalization*.

Unfortunately, the approaches and the tools offered so far by the SDC community do not fulfill the desiderata of a full-fledged solution needed by the processes of financial companies and organizations, like the Bank of Italy. First, to the best of our knowledge, all the existing SDC techniques are schema dependent and anonymization risk assessment and anonymization programs are tightly coupled to the dataset structure. Then, SDC techniques are only based on value statistics within the dataset to be anonymized and are not context-aware, while it is our experience that the risk of disclosure highly depends on the characteristics of the source and target databases [18] as well as the surrounding business information, e.g., availability of specific cross-linking data, even at tuple level. As a consequence, SDC techniques tend to fall short of accuracy in this respect. Although the anonymization techniques of SDC put into action interesting ideas and, in general, preserve statistical relevance of the datasets, to the best of our knowledge, all of them lack full explainability, unacceptable for financial organizations with strong accountability constraints. The lack of explainability prevents effective feedback-based adaptivity and the improvement of disclosure control proceeds by trial and error. Furthermore, all the existing tools tend to be not business friendly: they adopt a technology- and IT-dependent language (e.g., R libraries or Java), often lack clear semantics (typically only informally explained in the documentation), require adopters to have a technical background and are hardly extensible. Finally, such tools are data-scientist oriented libraries and, while showing good performance, do not have formal scalability guarantees.

Contribution. In this work we present Vada-Sa, a *reasoning-based* framework for financial data exchange with statistical confidentiality. It is based on our long-term experience in developing AI-enhanced data-driven solutions revolving around *logic-based reasoning*. In particular, this work builds on the Vadalog System [6], a state-of-the-art *reasoning system* leveraging the Vadalog language, a member of the Datalog$^{\pm}$ family [12], exhibiting very good characteristics of scalability and expressive power. In particular, we contribute as follows.

- We present a production-ready and fully engineered framework, Vada-Sa, for financial data exchange with confidential privacy, adopting a reasoning approach. The enterprise data to be shared, along with the metadata, are modeled as the *extensional component* of the reasoning process, whereas standard risk measures and anonymization methods are modeled as the *intensional component* of the process, i.e., a set of Vadalog rules. The activation of the rules upon the extensional component —i.e., the *reasoning process*— produces the *derived extensional component*, which is either a fully explained risk measure for a given dataset to be exchanged or its anonymized version.
- We show and discuss (and through Vada-Sa ship off-the-shelf) a number of risk measures and anonymization criteria and illustrate how they can be managed in Vadalog.
- We suggest that the surrounding *business context* relevant for accurate risk measures is awarely modeled within the intensional component in terms of Vadalog rules, which are at the same time *schema independent* w.r.t. the structure of the datasets. Although the framework targets financial data as a primary application, the techniques we present are general and can be applied in any context requiring statistical confidentiality.
- We envision that the SDC techniques can be used as a solid theoretical basis to craft a statistically preserving anonymization logic, yet, unlike existing approaches, we model in a purely *declarative* way in terms of Vadalog rules.
- We embrace a *user-delegation approach*, in the sense that by means of a semantically clear, fully declarative, non-technical and IT-independent language (i.e., characteristics that Vadalog embodies by design [6]), we delegate specific users to writing their own criteria and encoding the business knowledge, with cost and operational savings.
- In our framework, we inherit a set of benefits from logic-based reasoning. In particular, we refer to the pros of declarative approaches that, unlike procedural programming, relieve the users from the need to understand the internals of anonymization methods when adopting it. Full explainability is guaranteed by standard logic entailment semantics, enforced with chase-based procedures [20] embodied in Vada-Sa. Finally, the ideal balance between computational complexity and expressive power inherited from Vadalog, allows Vada-Sa to achieve very good scalability.
- We discuss an interesting set of real-world risk measures and anonymization criteria, implemented and operated in practice.

Overview. The remainder of the paper is organized as follows. In Section 2 we pursue the industrial setting at the Bank of Italy. In Section 3 we introduce the background about Vadalog. Section 4 presents the Vada-Sa framework and Section 5 shows it in action in relevant cases from the Bank of Italy. In Section 6 we discuss some related work and Section 7 concludes the paper.

2 INDUSTRIAL SETTING

The Bank of Italy has recently set up a *Research Data Center* (RDC).[1] At its core, there are a set of relational databases that store the *microdata*, i.e., the operational finest-grained data, from many core business applications such as the credit risk register, payment systems, balance of payments, banking supervision indicators, etc. The ultimate goal of RDC is sharing statistically relevant information with other cooperating institutions such as the National Statistical Office, other central banks, the European Central Bank, universities and research centers. While all these counter-parties operate within a "circle of trust", and can thus access the mentioned microdata, the identities of the involved entities, be they companies, banks or people, should remain of the sole responsibility (and therefore visibility) of the Bank of Italy, which is legally in charge of the respective processing.

The microdata that the RDC deals with regard different business processes and originate from multiple sources, usually external to the Bank of Italy. These data are collected with a variety of methods such as statistical surveys or data flows and are organized into several *microdata DBs*, by business domain. The RDC aims at including 65 microdata DBs with operational data from 1977 to 2020 and expected size of 30-50TB, with a 1TB/month growth. The RDC currently stores 14 microdata DBs, about families and individuals, firms, and historical data, including:

- Household income and wealth
- Household finance and consumption

[1] The RDC is part of the INEXDA initiative (http://www.inexda.org/) for the exchange of granular statistical data.

	I	Q	Q	Q	Q	Q	A	A	W
	Id	**Area**	**Sector**	**Employees**	**Residential Rev.**	**Export Rev.**	**Exp. to DE**	**Grwth 6mos**	**W**
1	612276	North	Public Service	50-200	0-30	0-30	30-60	2	230
2	737536	South	Commerce	201-1000	0-30	90+	0-30	-1	190
3	971906	Center	Commerce	1000+	0-30	30-60	0-30	4	70
4	589681	North	Textiles	1000+	90+	0-30	0-30	30	60
5	419410	North	Construction	1000+	90+	0-30	0-30	300	50
6	972915	North	Other	1000+	0-30	0-30	30-60	50	70
7	501118	North	Other	201-1000	60-90	90+	90+	-20	300
8	815363	North	Textiles	201-1000	60-90	30-60	90+	2	230
9	490065	South	Public Service	50-200	0-30	0-30	0-30	12	123
10	415487	South	Commerce	1000+	0-30	0-30	90+	3	145
11	399087	South	Commerce	50-200	30-60	0-30	30-60	2	70
12	170034	Center	Commerce	1000+	60-90	0-30	0-30	45	90
13	724905	Center	Construction	201-1000	0-30	30-60	0-30	2	200
14	554475	Center	Other	50-200	0-30	90+	0-30	0	104
15	946251	Center	Public Service	201-1000	30-60	90+	90+	150	30
16	581077	North	Textiles	50-200	0-30	60-90	30-60	-20	160
17	765562	South	Textiles	50-200	0-30	60-90	0-30	-7	200
18	154840	Center	Commerce	201-1000	0-30	60-90	0-30	4	220
19	600837	Center	Construction	50-200	0-30	60-90	0-30	20	190
20	220712	Center	Financial	1000+	30-60	60-90	30-60	-30	90

Figure 1: Microdata DB about inflation and growth.

- Financial literacy data
- Business outlook of industrial and service firms
- Italian housing market
- Inflation growth and expectations
- Historical archive of Italian credit.

Microdata DBs contain business data, including attributes that may disclose, directly or indirectly, the identity of the involved subjects; let us call these subjects *respondents*, by some abuse of the terminology adopted for statistical surveys. The risk for a tuple of a microdata DB to be associated (i.e., "linked") to the respective real-world identity of the respondent is named *risk of re-identification*. Indeed, the notion of re-identification revolves around the (realistic) assumption that an external data source containing all the identities of the respondents exists; let us call *identity oracle* such database. The challenge here consists in mitigating the risk that an attacker could be able to link the value of some attributes of a tuple of the microdata DB, with those of a single tuple (or a very small set thereof) of the identity oracle and therefore disclose the respondent's identity.

2.1 Setting Foundations

Let us frame our industrial context with the needed foundations.

Relational Foundations. Let $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{V}$ be disjoint countably infinite sets of *constants*, *(labelled) nulls* and (regular) *variables*, respectively. A *(*relational) schema $\mathbf{S}$ is a finite set of relation symbols (or predicates) with associated arity. A *term* is either a constant or variable. An *atom* over $\mathbf{S}$ is an expression of the form $R(\bar{v})$, where $R \in \mathbf{S}$ is of arity $n > 0$ and $\bar{v}$ is an n-tuple of terms. A *database instance* (or simply *database*) over $\mathbf{S}$ associates to each relation symbol in $\mathbf{S}$ a relation of the respective arity over the domain of constants and nulls. The members of relations are called *tuples*. By some abuse of notations, we sometimes use the terms tuple and fact interchangeably.

The Microdata DB and the Identity Oracle. A microdata DB is a relation of schema $M(\bar{i}, \bar{q}, \bar{a}, W)$, where $\bar{i}$ is an n-tuple of attributes defined as *direct identifiers*, $\bar{q}$ is an n-uple of *quasi-identifiers*, $\bar{a}$ is a set of *non-identifying* attributes and W is a *sampling weight*. An identity oracle is a relation of schema $O(\bar{i}', \bar{q}', I)$, where $\bar{i}'$ is a set of direct identifiers, $\bar{q}'$ is a set of quasi-identifiers and I is the *identity* of the respondent.

- *Direct identifiers* are attributes s.t. their values (of each single attribute, separately) allow to determine the identity of the respondent, that is, for a given tuple of M, the join between M and O on an attribute of $\bar{i}$ equated to an attribute of $\bar{i}'$ selects a single tuple from O and therefore the resulting tuple discloses the respondent's identity I. Observe that a direct identifier is a key attribute for O and it is assumed that $\bar{i} \subseteq \bar{i}'$. Examples of direct identifiers are the social security number, the Italian fiscal code, the driving licence number, etc.
- *Quasi-identifiers* are attributes s.t. the values of two or more of them, jointly, are likely to disclose the identity of the respondent, that is, for a given tuple of M, the join between M and O on two or more attributes of $\bar{q}$ equated to attributes of $\bar{q}'$ selects a small set of tuples of O and therefore likely discloses the respondent's identity I. In other terms, quasi-identifiers are features that in specific combinations are enough selective to endanger the respondent's confidentiality. This selectivity depends on the attribute (as some are intrinsically more specific) and, of course, on the combination of values, which can be more or less specific for a given context. For example, the joint use of age and address can be quite selective if we refer to a context of small dwellings, whereas gender and address would be less selective. On the other hand, occupation-gender is in general not very selective, whereas it can be extremely discriminating if we are referring to a context of a survey about gendered jobs in some country.
- *Non-identifying attributes* are those that do not fall in the two previous categories. These attributes are not critical because neither individually nor in combination with others, allow to disclose the identity of the respondent, i.e., re-identification is not possible. On the one hand, this can depend on an intrinsic scarce selectivity of the attribute like in the case of age in a given context, on the other hand, a non-identifying attribute can be even intrinsically identifying, yet its value certainly unknown to the identity oracle. This is the case, for instance, of internal system identifiers which are useless for re-identification.

- *Context and sampling weight.* We have touched on the notion of context when discussing quasi-identifiers, which are more or less selective depending on the domain of discourse. The context can be seen as a selection of tuples from O based on the domain of interest. For instance, if we were surveying the population of Milan, the only tuples of O referring to people living in Milan could be used to attempt re-identification of tuples of M, thus making it easier. The *sampling weight* accounts for the context by measuring the *representativeness* of a tuple t of M w.r.t. the entire context $\mathbb{C}$ to which M refers. In this sense, R is a sample from O, and W_t is the tuple sampling weight.
  There are different options for defining the sampling weight [7, 22]. The one we take inspiration from is the expected value of the number of entities having the same characteristics as t (according to a similarity function ϕ) in the sample distribution of O according to a given context $\mathbb{C}$. Given M, the weight W_t can be estimated for each tuple t from the posterior distribution of values for $\overline{q}$ among the tuples. Many options are also possible for ϕ and the simplest one just uses equality of quasi-identifiers attributes. Higher weights denote statistically relevant tuples, likely carrying scarcely selective attributes; lower weights denote statistically less relevant tuples (outliers, as a limit case), likely with highly selective attributes.
- *Identity.* The value of such attribute stands for some universally recognized representation of one respondent's identity.

In our experience with the Bank of Italy, the categorization of microdata DB attributes as direct, quasi- and non-identifiers as well as weight estimation is a hybrid process involving human experience-based evaluation, learning from training sets, and domain-based reasoning, as we shall see.

2.2 Towards a Reasoning Framework for Statistical Disclosure Control

With the depicted context, we can achieve a straightforward definition of *re-identification risk* as the probability $\rho_t = 1/W_t$ of re-identifying t given the value of all its quasi-identifiers $\overline{q}$. We can say that, in some sense, provided that O is an abstraction, the sampling weight W_t is an estimator for the cardinality of the join $|\sigma_t(M) \bowtie_{\overline{q}} O|$, where σ denotes the selection and $\bowtie$ the join.

However, re-identification risk is an upper bound for the real disclosure risk, in the assumption that all quasi-identifiers are known to a potential attacker. As a matter of fact, for a given tuple we may be interested in evaluating the risk only wrt a subset $\hat{q} \subset \overline{q}$ of quasi-identifiers, the ones we suppose the attacker is aware of or are more selective. Moreover, we may want to apply an arbitrary *risk weight* function λ, which takes as input W_t as well as the values for quasi-identifiers of t. Whence, the following definition of a general *statistical disclosure risk*:

$$\rho_{\hat{\mathbf{q}}} = 1/\lambda(\sigma_{\hat{q}=\hat{\mathbf{q}}}M) \tag{1}$$

The function λ computes an aggregate weight over the tuples selected by $\hat{\mathbf{q}}$ and generalizes many different risk measurement techniques, as we shall see, including the re-identification-based risk (for which $\lambda(\sigma_{\hat{q}=\hat{\mathbf{q}}}R) = \sum_{\sigma_{t\in\hat{q}=\hat{\mathbf{q}}}(R)} W_t$), but also *k-anonymity* [34], *individual risk* [7], and SUDA [19].

Use cases. Our industrial goal consists in: 1. evaluating the *statistical disclosure risk* and, 2. if unacceptable, take actions to counteract possible information disclosure, while preserving statistical significance (*anonymization*). The joint performance of the two mentioned actions is known as *statistical disclosure control* [18].

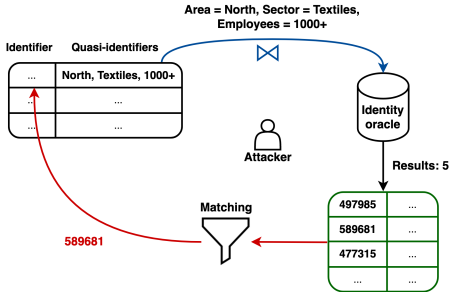


Figure 2: The attack strategy in action: by querying the identity oracle along Area, Sector and number of Employees, an attacker can narrow the search to few candidates and make a plausible guess about respondents' identity.

Figure 1 reports a fragment of a microdata DB of the Bank of Italy RDC, whose data derive from an *Inflation and Growth Survey*. This microdata DB shows the percentage growth in the last 6 months of Italian companies, spanning various sectors, with a different number of employees, different composition in revenue (residential viz. export) and a different percentage of export to Germany. The attributes of the microdata DB are the direct identifier $\overline{i} = \{Id\}$, where Id is a unique identifier for a company, the quasi-identifiers $\overline{q} = \{Area, Sector, Employees, ResidentialRevenue, ExportRevenue\}$, the non identifying attributes $\overline{a} = \{ExportToDE, Growth6mos\}$, and the weight $W = \{Weight\}$.

Re-identification risk is highest for tuple 15 (0.03) and lowest for tuple 7 (0.003). Extending to general (re-identification-based) statistical disclosure risk, we have that $\rho_{\hat{\mathbf{q}}}$ of a given tuple clearly coincides with re-identification risk if $\hat{q}$ includes the Id. It is also the case when $\hat{q}$ includes an n-uple of quasi-identifiers that happens to be unique. For example, tuple 4 is the only one located in the North, dealing in the Textiles sector, with more than 1000 employees; therefore, its re-identification and statistical disclosure risk coincide and amount to 0.016. Notice that its weight (60) witnesses the presence of multiple companies in the identity oracle having the same characteristics as tuple 4 according to the similarity function ϕ, e.g., the same/similar quasi-identifiers.

In another perspective, we are outlining a possible *attack strategy* to attempt re-identification of a given tuple t (Figure 2): 1. filter out a set of tuples C from O that match t on the values of attributes in $\overline{q}$; 2. choose the tuple $r \in C$ that best fits t w.r.t. the other attributes; 3. return r with an associated probability/score. To put the attack strategy into action, the entire toolbox from the *record linkage* literature can be adopted [13]. Efficient record linkage techniques typically operate in two steps: *blocking*, when restricting the cohort of candidate matches (step 1 of the attack strategy); *matching*, when evaluating the actual correspondences (step 2). Anonymization techniques aim at making blocking computationally expensive, by suppressing or modifying (as we shall see) selective values, which would make blocking effective restricting the cluster of candidate matches. With large clusters, exhaustive comparison is both computationally expensive, and yields an overly uncertain result, making the attack ineffective.

It is interesting to observe that the sampling weights can be used as a predictor of the effectiveness of a re-identification attack: tuples with higher weights in M will be in clusters with more candidates and thus less likely be identified, though statistically relevant; tuples with lower weights will be in smaller clusters and then will be more easy to re-identify. This gives an optimistic angle on the problem, as anonymization techniques can try to

operate on less representative tuples so as to increase overall confidentiality without hampering the statistical significance.

3 VADALOG REASONING

VADA-SA, the statistical disclosure control framework we introduce in this paper, is based on the VADALOG system, a state-of-the-art *logic-based reasoner* [6] whose core revolves around the VADALOG language, a member of the Datalog$^{\pm}$ family [12, 23]. The disclosure risk measurement techniques as well as the anonymization logic are expressed in VADALOG.

Datalog$^{\pm}$ generalizes Datalog with existential quantification in the rule conclusion, making it suitable for ontological reasoning. A *rule* is a first-order sentence of the form $\forall \bar{x} \forall \bar{y} (\varphi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z}\, \psi(\bar{x}, \bar{z}))$, where φ (the *body*) and ψ (the *head*) are conjunctions of atoms. For brevity, we omit universal quantifiers and denote conjunction by comma. As usual in this context, the semantics of a set of rules is operationally defined by the well-known CHASE procedure. Intuitively, the CHASE satisfies the rules by generating new head facts for bindings of the body, possibly introducing new variable symbols in the data, in the form of *labelled nulls*, in the presence of existentially quantified head variables [21].

The core of VADALOG is based on *Warded Datalog*$^{\pm}$ [6], a syntactic restriction to Datalog$^{\pm}$ that guarantees decidability and tractability in the presence of recursion and existential quantification. In terms of expressive power, Warded Datalog$^{\pm}$ captures full Datalog and `OWL 2` direct semantics entailment regime for `OWL 2 QL`. The language underpinnings are exploited by the reasoner to allow for efficient execution of reasoning tasks. VADALOG augments Warded Datalog$^{\pm}$ with supplementary features such as aggregation, algebraic operations, and stratified negation. As we shall see, VADALOG is sufficiently expressive to support our anonymization reasoning scenarios and comprises all the needed features such as joint use of full recursion, existential quantification and aggregation, to model propagation of the disclosure risk and anonymize values. These requirements are not met by the standard relational/SQL systems, which in particular offer inefficient or no support for recursion and existentials.

4 THE VADA-SA FRAMEWORK

While statistical disclosure control has traditionally followed a procedural approach, we propose a shift towards a fully declarative one, and look at state-of-the-art reasoners, leveraging our experience on VADALOG and Knowledge Graphs applied to different problems in the financial realm e.g., link prediction [2] as well as schema-independent approaches to model management [3]. The VADA-SA framework, whose architecture is sketched in Figure 3, lies on the following basic pillars:

- A structuring of the statistical disclosure control process in the form of an *anonymization cycle*.
- The construction of a VADALOG-based enterprise *Knowledge Base* (KB) encompassing the patterns and techniques for statistical disclosure risk assessment and anonymization as well as all the surrounding business knowledge to be leveraged.
- The formulation of risk assessment and anonymization phases in the form of *reasoning tasks* upon the KB. In such reasoning tasks, the *extensional component* comprises the microdata DB as well as their basic metadata, such as schema-level information. Much care is devoted to the *intensional component*, encoding reasoning rules for: attribute categorization, risk assessment and anonymization. The intensional component is at high level of abstraction, composed of pluggable VADALOG modules, some of which are provided off-the-shelf while others can be autonomously developed by business experts. The overall statistical disclosure control process is a reasoning task itself, which relies on the mentioned ones and adaptively chooses the actions to be performed. The *derived extensional component*, i.e., the results of the reasoning process, contains the outcome of risk analysis and the anonymized microdata DBs.

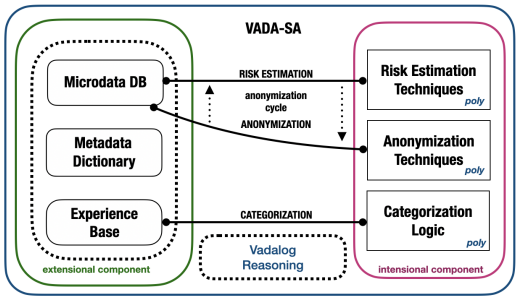


Figure 3: The VADA-SA architecture.

In this section, we first illustrate the anonymization cycle and the *metadata dictionary*, at the basis of our schema-independent approach (Section 4.1), we then focus on the evaluation of statistical disclosure risk (Section 4.2) and anonymization (Section 4.3). Finally, we show extensions and advanced applications involving complex business knowledge (Section 4.4).

4.1 The Anonymization Cycle and the Metadata Dictionary

When a microdata DB needs to be shared, it undergoes the *anonymization cycle* at the core of the VADA-SA architecture, shown in Figure 3. It consists of an iterative application of disclosure risk evaluation and anonymization until the risk is under a given threshold. Each iteration removes a minimum amount of information, and checks whether confidentiality requirements are fulfilled, in a statistics-preserving fashion.

In particular, *risk evaluation* takes as input a microdata DB. Based on its category, each attribute has a different treatment. Direct identifiers must not be disclosed and non-identifying attributes are not needed in the risk evaluation process, thus both are dropped. Quasi-identifiers and the sampling weight are used for disclosure risk estimation. Anonymization is activated until the disclosure risk is acceptable. In so doing, we aim at a trade-off between *statistical preservation* and disclosure risk, as captured by the threshold T, determined on the basis of user experience.

Metadata Dictionary and Attribute Categorization. In order to achieve schema and data independence, in VADA-SA we follow a *meta-level approach* and include a *metadata dictionary* in the KB. Facts of the form `MicroDB`(name), `Att`(microDB, name, description), `Category`(microDB,att,cat) are used to reason upon microdata DBs, their attributes and their categories, respectively. Figure 4 shows the portion of VADA-SA dictionary for the "I&G" (Inflation and Growth) microdata DB. Facts for `MicroDB` and `Attribute` are part of the extensional component and change when new microdata DBs are added into VADA-SA. Facts for `Category` are part of the derived extensional component: they are the product of a reasoning process that, for each microdata DB and each attribute, infers the most suitable category. In fact, before entering the anonymization cycle, the attributes of the microdata DB need to be categorized as identifiers, quasi-identifiers or non-identifying attributes, as we have seen in Section 2.1.

Algorithm 1 Attribute categorization

(1) $\text{Att}(M, A) \rightarrow \exists C\ \text{Cat}(M, A, C)$.
(2) $\text{Att}(M, A), \text{ExpBase}(A_1, C), A \sim A_1 \rightarrow \text{Cat}(M, A, C)$.
(3) $\text{Cat}(M, A, C) \rightarrow \text{ExpBase}(A, C)$.
(4) $\text{Cat}(M, A, C_1), \text{Cat}(M, A, C_2) \rightarrow C_1 = C_2$.

`Attribute`

Microdata DB	Attribute Name	Description
I&G	Id	Company Identifier
I&G	Area	Geographic Area
I&G	Sector	Product Sector
I&G	Employees	Num. of employees
I&G	Residential Rev.	Rev. from internal market
I&G	Export Rev.	Rev. from external market
I&G	Export to DE	Rev. from DE market
I&G	Growth	Rev. growth last 6 mths
I&G	Weight	Sampling Weight

`Category`

Microdata DB	Attribute Name	Category
I&G	Id	Identifier
I&G	Area	Quasi-identifier
I&G	Sector	Quasi-identifier
I&G	Employees	Quasi-identifier
I&G	Residential Rev.	Quasi-identifier
I&G	Export Rev.	Non-identifying
I&G	Export to DE	Quasi-identifier
I&G	Growth	Quasi-identifier
I&G	Weight	Sampling Weight

Figure 4: Metadata Dictionary: `Attribute` and `Category`.

Algorithm 1 shows the Vadalog program adopted for this purpose. It features a recursive application of experience: `ExpBase` is the extensional component and stores for an attribute name A, a known category C, according to available experts' knowledge. Assuming one category per attribute (Rule 1), if our attribute is sufficiently similar (according to a pluggable set of similarity functions or denoted by the $\sim$ symbol) to another attribute A_1 of the experience base for which the category is known, we borrow that category (Rule 2), and recursively feed the conclusion back into the experience base (Rule 3), so as to aid other decisions. Rule 4, technically an equality-generating dependency (EGD), guarantees that each attribute is assigned one single category. This Vada-Sa module lends itself to human-in-the-loop intervention in two points: when deciding whether to consolidate a decision of Rule 2 with Rule 3, as the user may consider a decision to be use-case specific, and when violations of EGD 4 arise, to allow for manual inspection of doubtful cases.

Anonymization Cycle. The interplay between evaluation of statistical disclosure risk and anonymization is at the core of our framework. Given the input microdata DB, the direct identifiers are removed and all the potentially harmful combinations of quasi-identifiers are evaluated to take countermeasures.

Algorithm 2 Anonymization cycle

(1) $\text{Val}(M, I, A, V), \text{Cat}(M, A, C), C \textit{ in } \{\text{Quasi-identifier}, \text{Weight}\}, \text{VSet} = \textit{munion}((A, V)) \rightarrow \text{Tuple}(M, I, \text{VSet})$.
(2) $\text{Tuple}(M, I, \text{VSet}), \#\text{risk}(I, R), R > T \rightarrow \#\text{anonymize}(\text{I})$.
(3) $\text{Tuple}(M, I, \text{VSet}), \#\text{risk}(I, R), R \leq T \rightarrow \text{Tuple}_A(M, I, \text{VSet})$.

The set of Vadalog rules of Algorithm 2 compactly represents this logic. Rule (1) creates `Tuple` facts for each tuple of the microdata DB M, identified by an artificial identifier I, and collects all the name-value pairs for quasi-identifiers and sample weights into the VSet variable. `Val` facts are part of the extensional component and store the value V for an attribute A of the microdata DB M. The identifiers of M are implicitly dropped. Observe that *munion* performs such aggregation, for each microdata DB M and VSet is a set-type variable. Whenever a specific tuple I violates a $[0, 1]$ risk threshold T, a fact `anonymize` is produced for I, in Rule 2. Both `risk` and `anonymize` are atoms defined in external libraries, in Vadalog (denoted by the "#" prefix). In particular, `risk` returns the risk R associated to a given tuple I; it is a compact form for the join "$\#\text{riskInput}(I), \#\text{riskOutput}(I, R)$", where `riskInput` is a fact triggering a Vadalog program producing facts of `riskOutput` for I. More simply, `anonymize` produces new facts for `Tuple`. This mechanism embeds a recursion on Rule 2, to anonymize tuples that still do not pass the risk evaluation. Only those facts for `Tuple` that pass the risk validation of Rule 3 are copied to Tuple_A, which can be considered anonymized.

The anonymization cycle in Algorithm 2 makes the approach *fully explainable* in the sense that each anonymization decision taken by Rule 2 is motivated by the specific binding of its body. It is also *preemptive* and *active*, in the sense that for each threshold violation, greedily applies a single anonymization step, at the same time minimizing the amount of suppressed statistical information. It is *schema independent*, as only atoms of the metadata dictionary are used and there is no specific reference to either schema or instance objects of the single microdata DBs. We shall see how specific values are bound to the attributes as a responsibility of `risk` and `anonymize` implementations. Finally, the algorithm offers multiple degrees of freedom: different risk and anonymization techniques can be used, and our `risk` and `anonymize` and polymorphic, in this sense; specific optimizations and execution heuristics can be adopted to choose which tuples to anonymize first (by controlling the activation order of Rule 2 against its possible bindings), and to choose which quasi-identifiers to anonymize first.

4.2 Statistical Disclosure Risk Estimation

Our `risk` atom in Algorithm 2 is polymorphic. Vada-Sa features a plug-in mechanism to opt for specific implementations at runtime. While the high-level characteristics of the Vadalog language allow to delegate users to specify their own risk logic, with extensive use of business knowledge, a number of techniques are provided off-the-shelf. In this section, we introduce the main risk disclosure evaluation techniques offered by Vada-Sa.

Re-identification-based. We start in Algorithm 3 with re-identification-based risk evaluation, that we have defined in Section 2.2.

Algorithm 3 Re-identification-based risk evaluation

(1) $\text{Tuple}(M, I, \text{VSet}), \text{riskInput}(I), \text{Cat}(M, W, \text{Weight}), R = 1/\textit{msum}(\text{VSet}[W], \langle I \rangle) \rightarrow \text{TupleA}(R, *\text{VSet}[\text{AnonSet}])$.
(2) $\text{Tuple}(M, I, \text{VSet}), \text{TupleA}(R, \text{VSet}*) \rightarrow \text{riskOutput}(I, R)$.

Whenever a tuple I needs to be evaluated (`riskInput` atom), in Rule 1, the name W of the weight atom is retrieved from the metadata dictionary and used to extract the weight value from VSet, with the *access* operator denoted by $[X]$, where X can be either a single attribute name or a collection thereof. Weights are summed (*msum*) and the risk score $1/R$ is computed. `TupleA` has

variable arity, and its terms are the computed risk R and the set of quasi-identifiers to group by when forming the summation. The expression $*$VSet[AnonSet] has the following meaning: the prefix operator "*" is *collection unpacking* and turns each element of the argument collection into a term of TupleA, essential for grouping along quasi-identifiers. Note that VSet is filtered by a set AnonSet of attribute names —the order is irrelevant in this context— that selects those that are considered of interest by business experts w.r.t. risk evaluation. Rule 2 finds those tuples I for which the risk has been computed and returns it. It uses the *packing operator*, denoted by the suffix operator "*", which packs a sequence of terms of TupleA into the set variable VSet. In this way, by joining on VSet, we can identify all the tuples to which the risk computation applies. We have already seen examples of re-identification-based risk estimation in Section 2.2.

k-anonymity is a commonly used threshold approximation of re-identification risk estimation [34]. For a given set of quasi-identifiers, whenever the number of occurrences is less than a fixed threshold k, it is considered dangerous; it is safe otherwise. For instance, in the microdata DB of Figure 1, considering the quasi-identifiers *Area* and *Sector*, we notice, e.g., that there is only one occurrence for "North" and "Public Service" (tuple 1). We say the set of pairs $\{\langle$Area, North$\rangle, \langle$Sector, Public Service$\rangle\}$ is a *sample unique* for tuple 1. The Vadalog reasoning rules for k-anonymity are reported in Algorithm 4.

Algorithm 4 k-anonymity

(1) $\text{Tuple}(M, I, \text{VSet}), \text{riskInput}(I),$
$R = \mathit{mcount}(\langle I \rangle) \rightarrow \text{TupleA}(R, *\text{VSet}[\text{AnonSet}]).$
(2) $\text{Tuple}(M, I, \text{VSet}), \text{TupleA}(R_1, \text{VSet}*)$
$R = \mathit{case}\ R_1 < k\ \mathit{then}\ 1\ \mathit{else}\ 0 \rightarrow \text{riskOutput}(I, R).$

Individual Risk. In the re-identification model the simplifying assumption is made that the sampling weight W_t corresponds to the frequency (number of occurrences) F_k of a given combination k of quasi-identifiers in the total population from which the microdata DB has been sampled; therefore we can compute the combination risk as $1/F_k$. Yet, frequencies F_k are unknown and in general different from W_t. A further inferential step is then required. The typical approach [7, 22, 38] is accounting for F_k in a Bayesian fashion, by considering the distribution of the population frequencies given the sample frequencies $F_k|f_{\hat{\mathbf{q}}}$ and obtaining $1/F_k$ as the posterior mean. In our setting, the sample frequency is the sample count in the microdata DB. Different assumptions can be made on the posterior distribution of $F_k|f_{\hat{\mathbf{q}}}$, with different techniques to accordingly estimate $\rho_{\hat{\mathbf{q}}}$. The one we adopt here is considering such distribution a negative binomial and thus we pose $\lambda = \sum W_t/f_{\hat{\mathbf{q}}}$ to estimate risk in Equation 1 of Section 2.2. Indeed, other distributions can be adopted. The individual risk estimation is formalized in Algorithm 5.

Algorithm 5 Individual risk

(1) $\text{Tuple}(M, I, \text{VSet}), \text{riskInput}(I), \text{Cat}(M, W, \text{Weight}),$
$F = \mathit{mcount}(\langle I \rangle), R = \mathit{msum}(\text{VSet}[W], \langle I \rangle) \rightarrow$
$\text{TupleA}(F/R, *\text{VSet}[\text{AnonSet}]).$
(2) $\text{Tuple}(M, I, \text{VSet}), \text{TupleA}(R, \text{VSet}*) \rightarrow \text{riskOutput}(I, R).$

While scanning through the tuples having a given combination VSet, used as a group-by key thanks to the unpacking operator, Rule 1 counts the occurrences of each combination (frequency) and sums the contributor weights. Facts for TupleA are produced only once all the contributors are available. The risk is then estimated for each combination and finally returned by Rule 2.

SUDA. With k-anonymity, we have introduced the concept of sample unique, i.e., a set of quasi-identifiers —name-value pair— that identify a tuple of a given microdata DB, i.e., they are unique. A sample unique is not the same as a database key, because it expresses a property that holds at tuple level and not at schema level. Alongside the schema-level distinction between superkey and key (a minimal superkey) in relational theory [1], here, at data level, we introduce the *minimal sample unique* (MSU) μ_t for a given tuple, that is a sample unique for which there exists no other sample unique μ'_t for the same tuple, s.t. $\mu'_t \subset \mu_t$. The *Special Unique Detection Algorithm* (SUDA) is a heuristic technique that estimates the statistical disclosure risk of a given tuple based on the size and the number of its MSUs.

Consider for example the set $\mu^1_{20} = \{\langle$Area, Center$\rangle, \langle$Sector, Financial$\rangle, \langle$Employees, 1000+$\rangle, \langle$Res. Rev., 30-60$\rangle\}$ for the microdata DB in Figure 1 for tuple 20. It is sample unique though not MSU, since the set $\mu^2_{20} = \{\langle$Sector, Financial$\rangle\}$ is sample unique and s.t. $\mu^2_{20} \subset \mu^1_{20}$. Moreover, μ^2_{20} is MSU. Similarly, $\mu^3_{20} = \{\langle$Employees, 1000+$\rangle, \langle$Res. Rev., 30-60$\rangle\}$ is another MSU. In total, tuple 20 has 2 MSUs.

Algorithm 6 encodes the Vada-Sa version of SUDA.

Algorithm 6 SUDA

(1) $\text{Tuple}(M, I, \text{VSet}), \text{riskInput}(I) \rightarrow \text{TupleI}(M, I, \text{VSet}).$
(2) $\text{TupleI}(M, I, \text{VSet}), \text{Cat}(M, A, \text{Quasi-identifier}),$
$A \in \text{VSet} \rightarrow \exists Z\ \text{Comb}(Z, I), \text{In}(A, Z).$
(3) $\text{Comb}(Z_1, I),\ \text{TupleI}(M, I, \text{VSet}), \text{Cat}(M, A, \text{Quasi-identifier}),$
$A \in \text{VSet}, \mathit{not}\ \text{In}(A, Z_1) \rightarrow$
$\exists Z\ \text{Comb}(Z, I), \text{InComb}(Z, Z_1), \text{In}(A, Z_1).$
(4) $\text{InComb}(X, Y), \text{In}(A, X) \rightarrow \text{In}(A, Y).$
(5) $\text{Comb}(Z, I), \text{In}(A, Z), \text{TupleI}(M, I, \text{VSet}),$
$\text{ASet} = \mathit{munion}(A) \rightarrow \text{TupleC}(I, *\text{VSet}[\text{ASet}]).$
(6) $\text{TupleC}(I, \text{VSet}*), \mathit{mcount}(\langle I \rangle) = 1 \rightarrow$
$\exists S\ \text{Su}(S, \text{VSet}), \text{HasSu}(I, S).$
(7) $\text{Su}(S, \text{VSet}), \text{HasSu}(I, S), \mathit{not}\ \text{HasSu}(I, S_1),$
$\text{Su}(S_1, \text{VSet}'), \text{VSet}' \subset \text{VSet} \rightarrow \text{MSU}(I, S).$
(8) $\text{TupleI}(M, I, \text{VSet}), \text{MSU}(I, S), \text{Su}(S, \text{VSet}),$
$R = \mathit{case}\ \mathit{size}(\mathit{VSet}) < k\ \mathit{then}\ 1\ \mathit{else}\ 0 \rightarrow \text{riskOutput}(I, R).$

After restricting the focus on input tuples (Rule 1), for each tuple we generate all the combinations of quasi-identifiers, first by introducing a combination Z for each of them (Rule 2), and then by constructing all the possible extensions that can be obtained by adding other quasi-identifiers (Rules 3 and 4). Then, for each combination of quasi-identifiers, generated by unpacking (Rule 5), we generate sample unique facts for Su, denoting those combinations that exactly identify one single tuple. The predicate HasSu is needed since every tuple I can have multiple sample unique sets, while the *mcount* aggregation needs to group by VSet. Rule 7 creates facts for MSU, filtering only those sample unique sets that are minimal. Finally, Rule 8 implements the logic to handle minimal sample unique sets. In this case, we evaluate the size of every MSU and if it is above a given threshold k, we consider the input tuple dangerous and thus return 1. The assumption here is that we cannot accept that the number of quasi-identifiers that can disclose the identity is too small. Clearly, more sophisticated checks could be implemented, possibly also including an overall evaluation of all the MSUs for a given tuple, for example by comparing the average size of MSUs against a threshold.

(a)

	I	Q	Q	Q	Q	
	Id	**Area**	**Sector**	**Employees**	**Residential Revenue**	**F**
1	099876	Roma	Textiles	1000+	0-30	1
2	765389	Roma	Commerce	1000+	0-30	2
3	231654	Roma	Commerce	1000+	0-30	2
4	097302	Roma	Financial	1000+	0-30	2
5	120967	Roma	Financial	1000+	0-30	2
6	232498	Milano	Construction	0-200	60-90	1
7	340901	Torino	Construction	0-200	60-90	1

(b)

	I	Q	Q	Q	Q	
	Id	**Area**	**Sector**	**Employees**	**Residential Revenue**	**F**
1	099876	Center	$\perp_1$	1000+	0-30	5
2	765389	Center	Commerce	1000+	0-30	3
3	231654	Center	Commerce	1000+	0-30	3
4	097302	Center	Financial	1000+	0-30	3
5	120967	Center	Financial	1000+	0-30	3
6	232498	North	Construction	0-200	60-90	2
7	340901	North	Construction	0-200	60-90	2

Figure 5: Local suppression and global recoding.

4.3 Smart Anonymization

In Algorithm 2 we have introduced the anonymization cycle, where Rules 2 and 3 show the interaction between risk estimation, with the main techniques introduced in Section 4.2, and anonymization, the object of this section. Tuples whose risk is considered over a given threshold T, produce facts for `anonymize`, which polymorphically triggers dedicated Vadalog programs: for each tuple I having statistical disclosure risk $R > T$, a new fact for `Tuple` is produced, with the same identifier I and statistical disclosure risk $R' < R$. The process continues recursively, until there is no tuple violating the threshold.

All the statistical disclosure evaluation techniques of Section 4.2 compute the risk associated to each tuple with monotonic aggregations, which play an important role here. In particular, all of them (e.g., *msum* in Algorithm 3, *mcount* in Algorithm 4, etc.) take as input the *aggregation contributor*, denoted by $\langle I \rangle$. According to the monotonic aggregation semantics [6], whenever two or more tuple tuples having the same value for the contributor I are aggregated (e.g., summed, counted, etc.) within the same group (defined by the bindings of head variables), only the tuple providing the least risk contribution is considered, while the others are neglected. This implies that, whenever a tuple I is replaced by a "more anonymous version", for example by suppressing a quasi-identifier, as we shall see, as the two are seen as the same contributor (they have the same value for I), only the anonymized one will be accounted for in the aggregation, so that more anonymized tuples incrementally replace the others and reduce risk, until convergence is achieved. We anonymize tuples with two main techniques: *introducing labelled nulls* to replace selective values, *applying a global recoding*.

Local Suppression with Labelled Nulls. Labelled nulls are a powerful tool from logic-based reasoning, which we effectively apply in the anonymization context. Consider the microdata DB in Figure 5a, where all the attributes are assumed to be quasi-identifiers, the sampling weight is omitted for simplicity, and the frequency of the n-uple of quasi-identifiers is showed on the right. For tuple 1 the set {⟨Area, Roma⟩, ⟨Sector, Textiles⟩, ⟨Employees, 1000+⟩, ⟨Resid. Rev., 0-30⟩} is sample unique. What if we replace the value "Textiles" for *Sector* with a labelled null $\perp_1$? As we are not aware of the underlying value of $\perp_1$, the combination of quasi-identifiers at hand may match with any among tuples 2-5, thus leading to a total frequency of 5. Likewise, tuples 2-5 see their frequency increased to 3. In total, by adding a single labelled null and hence introducing some degree of uncertainty, we have highly decreased the statistical disclosure risk of the microdata DB, as it can be seen in Figure 5b. In fact, a tuple containing one or more nulls may match with different tuples of the microdata DB, or even with none of them, depending on the specific assignment for those nulls.

Going back to what introduced in Section 2.2, in our framework in order to estimate the statistical disclosure risk, we need to compute $\lambda(\sigma_{\hat{q}=\hat{\mathbf{q}}}M)$ over a selection of the microdata DB M, based on an n-uple of values $\hat{\mathbf{q}}$ for quasi-identifiers. The risk estimation techniques of Section 4.2 apply λ to the entire microdata DB and the selection is implicit in the grouping performed by the aggregations, in the sense that, for each tuple, the aggregation forms the group by selecting only those tuples having the same values for quasi-identifiers (or subset of interest, thereof). So, $M(\bar{i}', \bar{q}', \ldots)$ is included in the selection induced by tuple $M(\bar{i}, \bar{q}, \ldots)$ iff $(q'_1, \ldots, q'_n) = (q_1, \ldots, q_n)$, and, by construction, the groups form a partition of the microdata DB. If we allow q_i to be a labelled null, a new semantics must be adopted to define whether $q_i = q'_i$ and thus form the aggregation groups.

The introduction of nulls raises non-trivial semantic issues when aggregations are involved, and theoretical work is still needed to achieve sound characterizations [25]. In Vada-Sa, for the construction of aggregation groups, we adopt a *null-tolerant semantics* inspired by the so-called *maybe-match* approach [14], and assume that $q_i =_\perp q'_i$ holds if: (i) q_i and q'_i have the same constant value, or (ii) either q_i or q'_i is a labelled null. Consequently, $(q'_1, \ldots, q'_n) =_\perp (q_1, \ldots, q_n)$ holds iff $q_i =_\perp q'_i$ holds for every $1 \leq i \leq n$. The $=_\perp$ relation is therefore used, instead of standard equality, to form groups. A tuple containing null quasi-identifiers, as a result of anonymization steps, is assigned to multiple aggregation groups (which do not partition the microdata DB anymore), increasing their cardinality and so anonymity.

We now have all the ingredients ready to encode local suppression, an anonymization method where quasi-identifiers are replaced by labelled nulls to reduce the statistical disclosure risk. The technique is expressed by Algorithm 7.

Algorithm 7 Local suppression

(1) $\text{Tuple}(M, I, \text{VSet}), \text{anonymize}(I), \text{Cat}(M, A, \text{Quasi-identifier}), \text{VSet}[A]$ *is not null* $\rightarrow \exists Z\ \text{Tuple}(M, I, (A, Z) \cup (\text{VSet} \setminus (A, \_)))$.

For a tuple I that needs to be anonymized, as witnessed by the predicate `anonymized`, for a not null quasi-identifier A, we generate a new tuple, where it is replaced by a labelled null Z.

Global Recoding. While local suppression introduces nulls, another technique to control statistical disclosure risk consists in decreasing the granularity of the values on the basis of domain knowledge. Consider again Figure 5a. Tuples 6 and 7 have the following sample unique sets, respectively: {⟨Area, Milano⟩ , ⟨Sector, Construction⟩},{⟨Area, Torino⟩, ⟨Sector, Construction⟩} and therefore have high disclosure risk. Besides the basic metadata dictionary we have seen in Section 4.1, the Vada-Sa KB contains knowledge about the attribute domains as well as the mutual relationship between their values. For instance, for the attribute *Area*, the KB comprises the following information:

Att(I&G, Area). TypeOf(Area, City). SubTypeOf(City, Region).
InstOf(Milano, City). InstOf(Torino, City).
InstOf(North, Region). IsA(Milano, North). IsA(Torino, North).

The *Area* attribute is known to be of `Type` "City", which in turn is a `SubtypeOf` "Region". Moreover we know that Milano and Torino are instances of cities and North is an instance of region. Finally, we now that both Milano and Torino are in the North. Similar knowledge is present for the entire geography.

Algorithm 8 Global recoding

(1) $\text{Tuple}(M, I, \text{VSet}), \text{anonymize}(I), \text{Cat}(M, A, \text{Quasi-identifier}), \text{TypeOf}(A, X), \text{subTypeOf}(X, Y), \text{isA}(\text{VSet}[A], Z), \text{TypeOf}(Z, Y) \rightarrow \text{Tuple}(M, I, (A, Z) \cup (\text{VSet} \setminus (A, \_)))$.

The logic for global recoding is in Algorithm 8. For a tuple that needs to be anonymized, we consider a quasi-identifier A. Based on its type, we climb the hierarchy up to its direct super-type Y. Then for the value $\text{VSet}[A]$ of A, we use the corresponding value Z of Y to replace $\text{VSet}[A]$. This form of suppression can be effectively applied to the entire microdata DB (and in this sense it is "global") and is inherently recursive as multiple hierarchical roll-ups may be needed to guarantee anonymity.

4.4 Enhancing Anonymization

We conclude the section by discussing two advanced topics: embedding of complex business knowledge, where we showcase the use of domain experience for *context aware* anonymization, and implementation of runtime heuristics, to maximize the statistical effectiveness of our approach.

Embedding complex business knowledge. The overall anonymization process can largely benefit from the surrounding business knowledge, an aspect often neglected by dedicated tools. Thanks to reasoning, we can inject business representations into different phases of Algorithm 2: in risk estimation modules, to craft ad-hoc methods; into anonymization techniques, e.g., to opt for specific values for global recoding, and so on. The setting we show here, motivated by our experience in the Bank of Italy with financial networks, consists of taking into account the relationships that exist between the respondents, say X and Y. It is in fact common that the statistical disclosure risk propagates along linked entities, e.g., companies or people, so that being able to re-identify one, makes it easier to re-identify others. In essence, all the linked entities of a given cluster, have the same disclosure risk, obtained as the probability that at least one entity of the cluster is re-identified: $1 - \prod_c (1 - \rho^c)$, where ρ^c is the risk of an entity, calculated with one of the techniques in Section 4.2. Here, along the lines of what usually done to estimate the risk of *households* and *hierarchical structures* [26], re-identification risk is interpreted as the re-identification probability.

Now, types of links that can be considered are arbitrarily complex: finding members of the same family, companies of the same company group are examples. The latter, e.g., could be encoded by the following Vadalog rules: (1) $\text{Own}(X, Y, W), W > 0.5 \rightarrow \text{rel}(X, Y)$. (2) $\text{rel}(X, Z), \text{Own}(Z, Y, W), msum(W, \langle Z \rangle) > 0.5 \rightarrow \text{rel}(X, Y)$. Clusters of companies ($\text{rel}(X, Y)$ holds where X and Y are in the same cluster) are defined by *company control relationships*: if X owns more than 50% of the shares of Y (Rule 1) or controls a set of companies Z that jointly own more than 50% of Y, than X controls Y and thus X and Y are in the same cluster.

Algorithm 9 shows an enhanced version of Algorithm 2, where the risk for a tuple I_2 is estimated as explained. Specifically, Rule 2 uses `#rel` (and we assume here $\text{rel}(X, X)$ holds) to compute the risk for I_1 as the combined risk of the entities in the same cluster. The aggregation *mprod* is the monotonic product, which considers, for each contributor I_2, the maximum contribution it provides, so as to account for the new less risky anonymized tuples produced by Rule 3, and eventually triggering Rule 4.

Runtime heuristics. We have seen how Vada-Sa operates incrementally and applies anonymization steps, only when tuples exhibit an overly high statistical disclosure risk. However, there

Algorithm 9 Enhanced anonymization cycle

(1) $\text{Val}(M, I, A, V), \text{Cat}(M, A, C), C \text{ in } \{\text{Quasi-identifier}, \text{Weight}\}, \text{VSet} = munion((A, V)) \rightarrow \text{Tuple}(M, I, \text{VSet})$.
(2) $\text{Cat}(M, A, C), C = \text{Identifier}, \text{Tuple}(M, I_1, \text{VSet}_1), \text{Tuple}(M, I_2, \text{VSet}_2), \#\text{rel}(\text{VSet}_1[A], \text{VSet}_2[A]), \#\text{risk}(I_1, R), R_{\text{clust}} = 1 - \#mprod(1 - R, \langle I_2 \rangle) \rightarrow \text{Risk}(\text{I}_1, \text{R}_{\text{clust}})$.
(3) $\text{Tuple}(M, I, \text{VSet}), \text{Risk}(I, R), R > T \rightarrow \#\text{anonymize}(\text{I})$.
(4) $\text{Tuple}(M, I, \text{VSet}), \text{Risk}(I, R), R \leq T \rightarrow \text{Tuple}_A(M, I, \text{VSet})$.

are still various open questions to be addressed, which correspond to specific degrees of freedom in anonymizing microdata. If there are two or more tuples that violate the risk threshold, which ones should be anonymized first? Moreover, if there are two or more quasi-identifiers of the same tuple, which one should be suppressed or recoded first?

As for the first question, in Vada-Sa, we adopt a *greedy* approach and choose to anonymize first the tuples that carry less statistical significance (namely, *data utility*), which can be estimated on the basis of the sampling weight. We exploit the so-called *routing strategies* [5] of the underlying Vadalog system to decide which bindings of the rule body to privilege when multiple possibilities arise. The approach here is quite intuitive: a *"less significant first"* strategy sorts the bindings of Rule 2 by risk and guides the anonymization accordingly.

The second question, namely the prioritization of quasi-identifiers, requires more care. We have seen in Algorithms 7 and 8 that either an existential or a higher-level domain value is used to replace quasi-identifiers and the specific attribute to consider is chosen as a consequence of the binding of $\text{Cat}(M, A, \text{Quasi-identifier})$. Also in this case, we can prioritize bindings by adopting a Vadalog routing strategy and the greedy approach. In particular, a *"most risky first"* strategy would first bind the rules against the attributes that affect more the tuple-level disclosure risk. So, in this case, the strategy itself would rely on a Vadalog program computing the risk, in order to take informed decisions. For instance, consider the problem of anonymizing tuple 1 of Figure 5a. Applying local suppression on *Sector* removes any sample unique of the tuple, which then occurs with frequency 5; instead, applying local suppression on *Area*, e.g., would leave the value "Textiles" for *Sector*, and would then require further local suppressions until such attribute is removed, with a consequential loss of data utility. In other terms, a greedy approach to local suppression or global recoding sustains the preservation of data utility.

5 EXPERIMENTS

Vada-Sa has been fully implemented and engineered in the Vadalog System. Towards a production application of the framework for the Research Data Center of the Bank of Italy, the system has been extensively experimented on real-world datasets from the Bank of Italy and synthetic ones to assess its *anonymization capability* (Section 5.1) and *scalability* (Section 5.2). The schema independent approach makes the framework general purpose and suitable for treating datasets in any domain.

Datasets. The microdata DBs used in the experimental analysis are reported in Figure 6. The *real-world* and *realistic datasets* derive from the *Inflation and Growth Survey* of the Bank of Italy, whose schema is shown in Figure 1. The *synthetic datasets* have been generated by fitting the real-world distribution (denoted by "W" in the figure) or by inducing specific unbalanced or very unbalanced distributions (denoted by "U" and "V"). Unbalanced

Dataset	No. Att.	No. Tuples	Dist.	Data
R6A4U	4	6k	U	Synth
R12A4U	4	12k	U	Synth
R25A4W	4	25k	W	Real-world
R25A4U	4	25k	U	Realistic
R25A4V	4	25k	V	Realistic
R50A4W	4	50k	W	Synth
R50A4U	4	50k	U	Synth
R50A5W	5	50k	W	Synth
R50A6W	6	50k	W	Synth
R50A8W	8	50k	W	Synth
R50A9W	9	50k	W	Synth
R100A4U	4	100k	U	Synth

Figure 6: Datasets used in the experimental settings.

distributions comprise many tuples with very selective combinations of quasi-identifiers, which exhibit high disclosure risk.

Hardware. We employed a memory-optimized virtual machine with 16 cores and 128 GB RAM on an Intel Xeon architecture.

5.1 Testing Anonymization Capability

We analyzed the capability of the system to detect tuples that need to be anonymized (i.e., the "risky" tuples).

Reduction of risk vs. loss of information. We applied the Vada-Sa anonymization cycle to the real-world dataset *R25A4W* with the *k-anonymity* risk evaluation technique (Section 4.2) and choosing a risk threshold $T = 0.5$. We employed *local suppression* anonymization (Section 4.3), with a *less significant first* runtime heuristic (Section 4.4). We varied the anonymity threshold k from 2 to 5. We adopted two metrics to evaluate the capability of the system to detect risky tuples: we counted the number of nulls injected by the local suppression as a result of risk evaluation, so analyzing how many values the system was able to erase (Figure 7a); we estimated the *loss of information* by weighing the number of erased values (i.e., the injected nulls) by the maximum total number of values, those of quasi-identifiers of the risky tuples w.r.t. T, that can be theoretically removed (Figure 7b) to satisfy the k-anonymity requirement. For both the measurements, we also evaluated the robustness of the approach, by applying the anonymization cycle to artificial but realistic datasets (*R25A4U* and *R25A4V*) having the same distribution of quasi-identifiers of *R25A4W*, but with an increased number of risky tuples.

The results in Figure 7a confirm what expected: when the k-anonymity threshold is increased, the anonymization cycle becomes less tolerant, and more redundancy is required to guarantee anonymity, therefore, in absolute terms, more and more labelled nulls need to be added to suppress specific values. We also observe that the number of needed nulls linearly grows with the tolerance threshold, as a consequence of an overall uniform distribution of values in the adopted combination of 4 quasi-identifiers. While an average real-world dataset requires less than 50 labelled nulls for 25k tuples with a 5-tuples tolerance threshold, more unbalanced versions require more, while confirming the trend. Figure 7b witnesses very good behaviour of Vada-Sa in terms of statistical preservation. For the real-world and the mildly unbalanced dataset, the loss of information is constantly below 20%, in particular between 12% (lower bound of *R25A4W*) and 17% (upper bound of *R25A4U*). For these two datasets, the constant trends show that when the number of risky tuples increases, the greedy approach succeeds in removing the values with a wider risk reduction effect. The loss of information for the very unbalanced dataset *R25A4V* is clearly higher, 37%, but it interestingly drops to 13% with less tolerant runs, because the high number of tuples that are considered risky on different combinations of values of quasi-identifiers, collapse in the k-anonymity comparison, when labelled nulls are introduced: so, while the number of nulls is high in absolute terms, the loss of information decreases. This result turns out to be an extremely positive guarantee of the anonymization capability of Vada-Sa.

Maybe-matching labelled nulls. In this experiment, we want to assess the effectiveness of the maybe-match semantics, which we use to compare labelled nulls with one another (and has been described in Section 4.3), as opposed to the standard semantics of labelled nulls, such as that adopted in chase-based procedures (e.g., Skolem chase [11]). According to the standard semantics, for a quasi-identifier q_i, we have that $q_i =_\perp q_i'$ holds if: (i) q_i and q_i' have the same constant value, or (ii) both q_i and q_i' are labelled with the same null symbol. We plugged this semantics into Vada-Sa, used the same real-world and realistic datasets of Figure 7b and report the number of injected nulls by k-anonymity threshold in Figure 7c. Also here, the risk threshold $T = 0.5$ has been used. The figure highlights the proliferation of symbols (the red lines) that takes place with the standard semantics, which is in fact unusable in this setting. By contrast, the probabilistic interpretation of nulls we foster, minimizes the number of labelled nulls (the light-blue lines, whose zoomed version is in Figure 7a).

Using business knowledge. We show the results of anonymization in a real-world setting where anonymization cycle is complemented with a set of Vadalog rules that produce derived extensional knowledge about control relationships between companies. The rules and the setting have been presented in detail in Section 4.4. For the test, we adopt the real-world dataset *R25A4W* and its tweaked unbalanced versions, *R25A4U* and *R25A4V*. We anonymize each of the datasets by estimating the risk with k-anonymity with $k = 2$ and $T = 0.5$. We measure the number of nulls injected by local suppression in 5 settings, with increasing number of inferred control relationships, from 0 to 400.

The results are shown in Figure 7d. With all the datasets, the number of injected nulls grows with the number of relationships between entities, which induce bigger and more risky clusters. The three distributions of the quasi-identifier values differently interact with the derived relationships: the more unbalanced the dataset is, the more tuples will be affected by the propagation of risk of the outliers, resulting into a globally risky dataset, to be severely anonymized. In real-world tests, relationships disclose many cases that deserve anonymization (from 9 in the case of 100 relationships to 38 for 400), while the propagation effect is maximized in the *R25A4V* dataset with an upper bound of 323 injected nulls for 300 relationships.

5.2 Testing Scalability

Given the characteristics of the data at hand to be anonymized, we need to make sure that our approach scales well. Although the anonymization cycle, risk estimation and anonymization of Vada-Sa are expressed in Vadalog, where reasoning is PTIME in data complexity [6], here we want to investigate on the specific runtime of the system in different settings.

By dataset size. We tested the scalability of Vada-Sa by increasing volumes, with 4 synthetic datasets (from *R6A4U* to *R100A4U*), unbalanced and having a high number of risky tuples. We measured the elapsed time for the entire anonymization cycle and also pointed out the sole risk estimation component, with 3 different risk estimation techniques (*individual risk*, *k-anonymity*, *SUDA*). We used $k = 2$ for k-anonymity, 3 as the MSU threshold

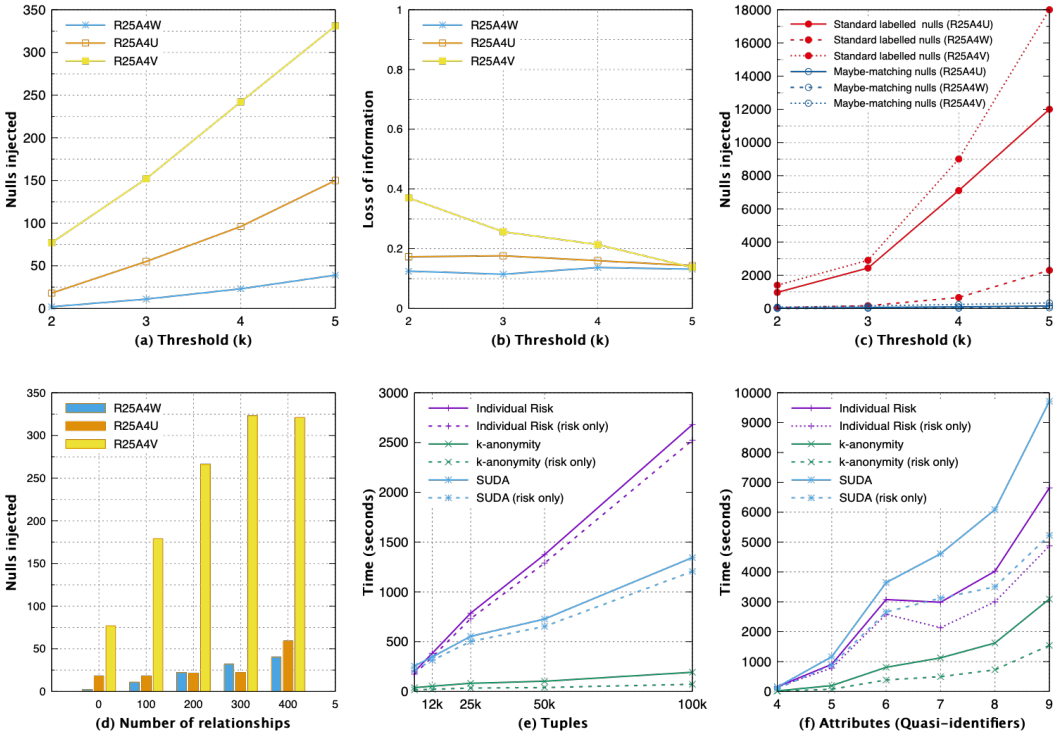


Figure 7: (a) Number of nulls injected by k-anonymity threshold. (b) Information loss by k-anonymity threshold. (c) Number of nulls injected with maybe-matching viz. standard labelled null semantics. (d) Number of nulls injected by increasing number of relationships in settings with explicit modeling of business knowledge. (e) Execution time by dataset size and risk estimation technique. (f) Execution time by number of quasi-identifiers and risk estimation technique.

for SUDA (see Section 4.3) and $T = 0.5$. We executed each run 5 times and averaged the measurements, for a total of 60 runs.

The results are shown in Figure 7e. All the three groups of trends confirm that the risk estimation component (dotted lines) dominate the elapsed time. This is reasonably expected, as the convergence of our anonymization cycle depends on a positive evaluation of risk estimation, which is then the bottleneck. The linear trend confirms the applicability of the approach. In particular, k-anonymity exhibits a very good behaviour, with elapsed time between 6 and 192 seconds for $100k$ tuples. The limited cost of estimation can be ascribed to the adoption of monotonic aggregations, which adopt incremental updates and need not be recomputed from time to time. Whilst in Algorithm 5 we have made a simple assumption to estimate the risk from the posterior distribution $F_k|f_{\hat{q}}$ (which would have led to elapsed times similar to those for k-anonymity), for this experiment we plugged into Vada-Sa an off-the-shelf statistical library and sampled from the actual negative binomial distribution. The costly trend is motivated by the interaction overhead between the native Vadalog component and the library. The trend for SUDA is less than linear, with, e.g., 727 seconds for $50k$ and 1344 seconds for $100k$ tuples since the potential blowup on the number of examined combinations of quasi-identifiers is controlled by the Vadalog optimizations.

By number of quasi-identifiers. To investigate more the dependence of performance on the number of quasi-identifiers, we stressed Vada-Sa by anonymizing 6 datasets *R50A4W-R50A9W*, so with increasing number of attributes and fixed number of tuples, $50k$, and real-world-like distribution. We used the same thresholds for k-anonymity, SUDA, and T. We measured elapsed time and the risk estimation component. We executed each run 5 times averaging the results, for a total of 90 runs.

Figure 7f reports the results. As expected, individual risk and k-anonymity are only marginally affected by the increased number of quasi-identifiers, as they do not consider all the combinations with at most k attributes, but only those with exactly k. Instead, we may expect a much worse trend for SUDA, where for each tuple, all the combinations of at most k attributes are inspected to detect potential MSU. Remarkably, no combinatorial blowup appears in the figure, witnessing a very effective behaviour of the Vadalog execution optimization: while the activation of Rules 2-5 of Algorithm 6 could in theory cause a blowup w.r.t. k, it does not happen in practice because the greedy activation of Rule 7 performed by Vadalog to detect the MSUs preempts the generation of redundant combinations of quasi-identifiers.

6 RELATED WORK

Statistical disclosure control is a broad topic to which many have contributed, especially from the Statistics community, whose work can be considered related to ours.

The concept of *Sample Uniqueness* (SU) to measuring the risk of data disclosure was introduced by Skinner [35], while *k-anonymity*, was first presented by Sweeney [37], along with the first methods of anonymization by generalization (our *global recoding*) and *local suppression*. The measure of *individual risk* in our contribution is inspired by the work of Benedetti and Franconi [7] who proposed to compute the risk of data disclosure with the sampling weights of data records.

The topic of data anonymization is related to the area of *differential privacy* [17], where an interesting concept may be adopted in our approach so as to develop a new family of risk measures,

based on the idea that an individual's privacy may be violated even knowing the absence of the individual from the microdata. Investigating such direction will be matter of future work.

While the foundations of our work are set in the theory of statistical disclosure control, our contribution is concerned with providing an industrial production ready solution for the Bank of Italy, conveying a set of properties that derive from a fully declarative reasoning approach. In this sense, we combine our experience in logic-based reasoning [6] and schema-independent solutions to model management problems [3]. None of the existing dedicated software solutions for statistical disclosure control offers the mentioned set of properties. The software pack ARGUS [27] aims at local suppression and coding, as does the Datafly system [36]. Manning et al. introduce SUDA2 (Special Unique Detection Algorithm) [29], whose objective is to detect the risk in certain unique combinations of variables. Recently, the *R* package *sdcMicro* has implemented many of the risk measures and anonymization approaches of our interest [9]. Likewise, ARX is a solution for data anonymization that has been proposed as a practical approach to Statistical disclosure control [33]. A comprehensive survey of the statistical approaches has been provided by Matthews and Harel [30]. Recent work on the risk of information disclosure in linked data, and, more in general, ontology-based data, has formalized the problem and defined its logical foundations [8], with an interest in the concept of linkage safety in RDF graphs [24]; a declarative framework for linked data anonymization has also been proposed [15]. The problem of preserving privacy in data exchange has been analyzed also in the context of information integration systems [31] where a practical solution is represented by *MapRepair* [10] and in the cryptography community, with homomorphic encryption [28].

In the AI literature, statistical disclosure control has been mostly considered within machine learning [16] and deep learning approaches [4]. Yet, they have a different focus and aim at generating anonymized clones of existing datasets while respecting the original statistical properties. An interesting deductive proposal by Øhrn and Ohno-Machado uses Boolean reasoning for data anonymization in databases [32], which however remains purely theoretical and just considers the combinatorial aspect.

7 CONCLUSION

In this paper, we presented Vada-Sa, a declarative statistical disclosure control framework. We demonstrated the anonymization workflow, metadata dictionary, and statistical disclosure risk estimation. Utilizing these components, we introduced the anonymization cycle. To maximize the statistical effectiveness of our approach, we also presented two enhancements, namely embedding of complex business knowledge and runtime heuristics. We validated the approach on real-world central bank data. As future work, we plan to further enhance the framework, and test it in a variety of other real-world scenarios.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[2] Paolo Atzeni, Luigi Bellomarini, Michela Iezzi, Emanuel Sallinger, and Adriano Vlad. 2020. Weaving Enterprise Knowledge Graphs: The Case of Company Ownership Graphs.. In *EDBT*. 555–566.

[3] Paolo Atzeni, Luigi Bellomarini, Paolo Papotti, and Riccardo Torlone. 2019. Meta-mappings for schema mapping reuse. *PVLDB* (2019).

[4] Brett K. Beaulieu-Jones, Zhiwei Steven Wu, Chris Williams, Ran Lee, Sanjeev P. Bhavnani, James Brian Byrd, and Casey S. Greene. 2019. Privacy-Preserving Generative Deep Neural Networks Support Clinical Data Sharing. *Circ Cardiovasc Qual Outcomes* 12, 7 (07 2019).

[5] Luigi Bellomarini, Davide Benedetto, Georg Gottlob, and Emanuel Sallinger. 2020. Vadalog: A modern architecture for automated reasoning with large knowledge graphs. *Inf. Syst.* (2020), 101528.

[6] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *PVLDB* 11, 9 (2018), 975–987.

[7] Roberto Benedetti and Luisa Franconi. 1998. Statistical and technological solutions for controlled data dissemination. In *Pre-proceedings of New Techniques and Technologies for Statistics*, Vol. 1. 225–232.

[8] Michael Benedikt, Bernardo Cuenca Grau, and Egor V Kostylev. 2018. Logical foundations of information disclosure in ontology-based data integration. *Artificial Intelligence* 262 (2018), 52–95.

[9] Thijs Benschop, Cathrine Machingauta, and Matthew Welch. 2019. Statistical Disclosure Control: A Practice Guide. (2019).

[10] Angela Bonifati, Ugo Comignani, and Efthymia Tsamoura. 2019. MapRepair: Mapping and Repairing under Policy Views. In *Proceedings of the 2019 International Conference on Management of Data*. 1873–1876.

[11] Andrea Calì, Georg Gottlob, and Michael Kifer. 2013. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. *JAIR* 48 (2013), 115–174.

[12] Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, and Andreas Pieris. 2011. Datalog+/-: A Family of Languages for Ontology Querying. In *Datalog Reloaded*.

[13] Peter Christen. 2012. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer.

[14] Margareta Ciglic, Johann Eder, and Christian Koncilia. 2014. k-anonymity of microdata with NULL values. In *International Conference on Database and Expert Systems Applications*. Springer, 328–342.

[15] Rémy Delanaux, Angela Bonifati, Marie-Christine Rousset, and Romuald Thion. 2018. Query-based linked data anonymization. In *International Semantic Web Conference*. Springer, 530–546.

[16] Jörg Drechsler and Jerome Reiter. 2011. An empirical evaluation of easily implemented, nonparametric methods for generating synthetic datasets. *Computational Statistics & Data Analysis* 55 (2011), 3232–3243.

[17] Cynthia Dwork. 2006. Differential Privacy. In *Automata, Languages and Programming*, Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12.

[18] Mark Elliot and Josep Domingo-Ferrer. 2018. The future of statistical disclosure control. *The National Statistician's Quality Review* (2018).

[19] Mark J Elliot, Anna M Manning, and Rupert W Ford. 2002. A computational algorithm for handling the special uniques problem. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 05 (2002), 493–509.

[20] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2003. Data Exchange: Semantics and Query Answering. In *ICDT*.

[21] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data exchange: Semantics and query answering. *TCS* 336, 1 (2005), 89–124.

[22] Luisa Franconi and Silvia Polettini. 2004. Individual risk estimation in μ-Argus: A review. In *Int. Workshop on Privacy in Statistical Databases*. 262–272.

[23] Georg Gottlob, Thomas Lukasiewicz, and Andreas Pieris. 2014. Datalog+/-: Questions and Answers. In *KR*.

[24] Bernardo Cuenca Grau and Egor V Kostylev. 2019. Logical foundations of linked data anonymisation. *J. of AI Research* 64 (2019), 253–314.

[25] Paolo Guagliardo and Leonid Libkin. 2019. On the Codd semantics of SQL nulls. *Inf. Syst.* 86 (2019), 46–60.

[26] Anco Hundepool, Josep Domingo-Ferrer, Luisa Franconi, Sarah Giessing, Eric Schulte Nordholt, Keith Spicer, and Peter-Paul De Wolf. 2012. *Statistical disclosure control*. John Wiley & Sons.

[27] Anco Hundepool, A Van de Wetering, Ramya Ramaswamy, Peter-Paul de Wolf, Sarah Giessing, Matteo Fischetti, Juan-José Salazar, Jordi Castro, and Philip Lowthian. 2005. τ-argus user's manual, version 3.3. *Statistics Netherlands, Voorburg, The Netherlands* (2005).

[28] Michela Iezzi. 2020. Practical Privacy-Preserving Data Science With Homomorphic Encryption: An Overview. *CoRR* abs/2011.06820 (2020).

[29] Anna M. Manning, David J. Haglin, and John A. Keane. 2008. A recursive search algorithm for statistical disclosure assessment. *Data Mining and Knowledge Discovery* 16, 2 (01 Apr 2008), 165–196.

[30] Gregory Matthews and Ofer Harel. 2011. Data confidentiality: A review of methods for statistical disclosure limitation and methods for assessing privacy. *Stat. Surv.* 5 (01 2011).

[31] Alan Nash and Alin Deutsch. 2007. Privacy in GLAV information integration. In *International Conference on Database Theory*. Springer, 89–103.

[32] Aleksander Øhrn and Lucila Ohno-Machado. 1999. Using Boolean reasoning to anonymize databases. *Artificial intelligence in medicine* 15 3 (1999), 235–54.

[33] Fabian Prasser and Florian Kohlmayer. 2015. Putting statistical disclosure control into practice: The ARX data anonymization tool. In *Medical Data Privacy Handbook*. Springer, 111–148.

[34] Pierangela Samarati and Latanya Sweeney. 1998. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. (1998).

[35] Chris Skinner, Catherine Marsh, Stan Openshaw, and Colin Wymer. 1994. Disclosure control for census microdata. *JOS* 10, 1 (1994), 31–51.

[36] Latanya Sweeney. 1997. Guaranteeing anonymity when sharing medical data, the Datafly System. *Proc. AMIA Fall Symposium* (1997), 51—55.

[37] Latanya Sweeney. 2002. Achieving k-anonymity privacy protection using generalization and suppression. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 05 (2002), 571–588.

[38] Matthias Templ, Alexander Kowarik, and Bernhard Meindl. 2013. sdcMicro: Statistical Disclosure Control methods for the generation of public-and scientific-use files. *Manual and Package* (2013).

Industrial Paper

Generating Realistic Test Datasets for Duplicate Detection at Scale Using Historical Voter Data

Fabian Panse
panse@informatik.uni-hamburg.de
Universität Hamburg
Hamburg, Germany

André Düjon
1duejon@informatik.uni-hamburg.de
Universität Hamburg
Hamburg, Germany

Wolfram Wingerath
wolfram.wingerath@baqend.com
Baqend
Hamburg, Germany

Benjamin Wollmer
wollmer@informatik.uni-hamburg.de
Universität Hamburg
Hamburg, Germany

ABSTRACT

The detection of duplicates is an essential task in data cleaning and integration and has steadily gained importance especially for researchers and practitioners that need to process and integrate large volumes of potentially unclean data on a daily basis. To evaluate the quality and performance of duplicate detection algorithms, labeled test data are required that provide information on the contained duplicates. Current approaches for generating test data, however, are either not scalable (and therefore limited to small datasets) or not able to generate realistic data values and errors, especially outdated values. In this paper, we propose a scheme for generating test datasets that addresses both these issues and present a test dataset generated with it. Our approach relies on using historical data from the North Carolina voter register which (1) is realistic as it contains actual voter data and (2) facilitates generating realistic duplicates through the fact that current data values were collected at every election through manually filled out applications. The generated test dataset comprises more than 120 million records with up to 90 attribute values each. To the best of our knowledge, we are the first who provide realistic test data for duplicate detection at this scale.

1 INTRODUCTION

Duplicates are data records (e.g., tuples in the relational case) that refer to the same real-world object. They can result from errors in data management, but also occur because separately developed data sources overlap in their universes of discourse (e.g., many actors and movies are stored in both IMDB[1] and TMDb[2]). The detection of duplicates is an important task in data cleaning [12, 16] and integration [8, 9]. Detecting duplicates is quite simple when they are exact, i.e. they agree in all of their values. However, it can be extremely difficult if some of their values disagree due to typos, phonetic or transformation errors, heterogeneous forms of presentation as well as missing or outdated values [14].

The challenge of detecting such so-called *fuzzy* duplicates has opened up its own field of research and has since been studied intensively [4, 7, 23, 31]. However, the best approach to find them strongly depends on (i) the considered domain (e.g., movies, persons, or proteins), (ii) the characteristics of the given data (e.g., volume, data model and heterogeneity), and (iii) the quality and cost requirements of the user (e.g., good results vs. short runtimes and recall vs. precision). Due to the resulting diversity of use cases, none of the existing algorithms has turned out to be a generally applicable and superior solution. Instead, in every use case, it remains a difficult (and expensive) task to choose and configure them so that they provide adequate results.

Such a configuration process requires the evaluation and comparison of different algorithms and parameter settings. This in turn requires test datasets that do not only provide a *gold standard* (a.k.a. *ground truth*) labeling the dataset's duplicates [21], but resemble the required real-life properties as well as possible. Current approaches to test data generation either (i) struggle with the generation of realistic data values and errors (especially outdated values), (ii) cannot guarantee the soundness of the gold standard[3], or (iii) scale badly and thus can only be used to generate small datasets. However, realistic values and errors as well as correctly labeled duplicates are an important prerequisite for a test dataset. Moreover, in times of big data many duplicate detection algorithms focus on scalability (e.g., [13, 17, 30]) so that an evaluation of their key functionalities requires large test datasets with millions of records.

Using a historical dataset to generate test data seems to be a straightforward solution to some of the aforementioned problems, because the mapping between records and real-world objects is part of the data so that the duplicates are already labeled. Thus, it scales much better than, for example, labeling the duplicates in an unclean dataset manually. In addition, historical datasets are perfectly suited to generate outdated values, because these values are an inherent part of the data. One of these historical datasets is provided by the State of North Carolina (short NC) [26]. This dataset contains information on voters registered to the individual elections and – at the time of our study – consisted of 45 snapshots covering a time period of 16 years with a total number of over 500 million records and a large schema with 90 attributes. These numbers make it a perfect candidate to evaluate the suitability of the aforementioned idea, because the large number of records allows us to generate a test dataset of large size and the large time span provides us many outdated values (even more than one for the same object property). Furthermore, since voters often have to re-register at regular intervals by manually filled out forms[4], the registration data contain typos, values confused between attributes, heterogeneous forms of presentation and missing values which makes this dataset particularly useful for

[1] Internet Movie Database: https://www.imdb.com
[2] The Movie Database: https://www.themoviedb.org

[3] To clearly distinguish between errors in the duplicate labels and the operational data, we use the term *soundness* w.r.t. the correctness of the gold standard.
[4] https://dl.ncsbe.gov/?prefix=Voter_Registration/

the generation of fuzzy duplicates. Finally, its large size gives us the opportunity to customize the test data to different user requirements by selecting a suitable subset of all records (the more data available, the more flexible the selection process). However, the big amount of redundant data as well as the ongoing publication of new snapshots also pose some challenges to the generation process making it a non-trivial task.

The contributions of this paper can be summarized as:

(i) A comprehensive list of desiderata for test datasets for duplicate detection.
(ii) An approach for generating and storing test data based on the historical voter register from North Carolina.
(iii) A realistic test dataset generated with our approach.
(iv) An extensive experimental evaluation to analyze the quality and prove the usability of the generated test dataset.

We provide the generated dataset to other researchers[5]. It will help them to evaluate their algorithms (such as runtime behavior or robustness against a varying number of data errors) and to compare them with those of other research projects. It is particularly valuable to the research community through a combination of properties that is unique to the best of our knowledge:

- It contains more than 120 million records and 640 million duplicate pairs making it suitable to evaluate duplicate detection algorithms at scale,
- it contains real-life errors of various types including typos, abbreviations, phonetic errors and outdated values,
- its large size qualifies it to customize the test data to different user requirements without losing necessary volume,
- it provides precalculated plausibility and heterogeneity scores, which support the user to remove (or repair) potentially unsound duplicate clusters and adapt the datas' heterogeneity to her own requirements, and
- it provides meta information that allows the user to reproduce experiments using previous versions of this dataset.

The rest of this paper is structured as follows: In Section 2, we describe the input to our study, i.e. the voter register from North Carolina. Thereafter, in Section 3, we discuss several aspects affecting the test datas' quality, usability and reproducibility. In Sections 4 and 5, we describe our approach for using the historical voter data for test data generation. In Section 6 we present an experimental study that evaluates the quality and usability of the generated test dataset. Finally, we discuss related work in Section 7 before we conclude the paper and give an outlook on future research in Section 8.

2 NORTH CAROLINA VOTER REGISTER

The voter register from North Carolina was created and is still maintained by the *North Carolina State Board of Elections*[6] according to the *Help America Vote Act* (HAVA) of 2002. The provided voter records are considered public information per NC General Statutes (§132-1, §163-82.10) [1, 25], but do not include dates of birth, driver's license numbers and social security numbers because they are confidential under state law [1, 20, 26].

In addition to current data, the register provides a voter history in the form of a series of snapshots [24]. The first publicly available snapshot is from 2005-11-25. New snapshots were (and still are) created at every New Year's Day and the date of every election (general, primary and municipal) [26]. At the time of our study, the register contained 45 snapshots. The schema of these snapshots evolved over time, but was consistent for the last 41 snapshots. Since the first four snapshots are missing necessary information to clearly identify a voter, we excluded them from our study. The characteristics of the remaining snapshots are presented (in an aggregated form) in Table 1. The whole voter history contains 522,463,029 records representing a total of 13,569,512 distinct persons.

Table 1: Overview of the snapshots included in this study

	#snap-	#total	#new		rate of new	
year	shots	records	records	objects	records	objects
2008	1	9.7 M	9.7 M	9.4 M	100%	96.8%
2009	1	9.7 M	0.7 M	37 K	6.8%	5.6%
2010	2	20.2 M	13.1 M	189 K	64.9%	1.4%
2011	1	10.3 M	2.3 M	225 K	22.2%	9.9%
2012	4	41.8 M	19.9 M	820 K	47.6%	4.1%
2013	1	11.4 M	11.1 M	41 K	97.1%	0.4%
2014	4	47.3 M	7.5 M	432 K	15.8%	5.8%
2015	4	49.0 M	6.6 M	223 K	13.5%	3.4%
2016	4	50.9 M	7.7 M	587 K	15.1%	7.6%
2017	4	54.1 M	3.6 M	245 K	6.7%	6.7%
2018	3	41.7 M	23.7 M	374 K	56.9%	1.6%
2019	7	99.8 M	5.5 M	354 K	5.5%	6.5%
2020	4	60.8 M	8.0 M	596 K	13.1%	7.4%
2021	1	15.9 M	0.8 M	62 K	5.1%	7.6%
total	41	522.5 M	120.8 M	13.57 M	23.1%	11.2%

M = million, K = thousand

Each snapshot corresponds to a large tab-separated values (TSV) file. As it turned out during data profiling, these files are formatted differently. While the older files (if not updated later) are in UTF-8, the newer files are in UTF-16. Since none of the provided attributes is expected to contain characters that are not part of the UTF-8 character set, we converted all files to UTF-8 before importing them into our dataset. Here it is important to note that occasional conversion errors do not spoil our test dataset, since they also happen in real-life, as long as they do not corrupt the correct mapping between records and objects required for the gold standard (i.e., they do not concern the NCIDs).

Every record in the snapshot files specifies an entry to the voter register and consists of 90 attributes. We grouped these attributes into four semantic categories:

- personal information (38 attributes) such as names, age, address data, phone number, race code and sex,
- information on the districts the voter is registered in (38 attributes), such as school, water and fire district,
- information on the voter that is closely related to the election she is registered to (11 attributes), such as voter status and registration date, and
- meta data for administrating the snapshots and identifying voters/records within them (3 attributes), which are the NCID as well as the snapshot and load date.

The NCID is a unique number for each voter currently or previously registered in North Carolina. A voter's NCID will follow him from one county to another when she migrates within the state of NC. Thus, the NCID can be used to uniquely identify the individual voters and therefore can serve as an object-id. To our surprise, we discovered that in every snapshot many voters are represented by more than one record. A closer look revealed that at most only one of them has not the voter status *removed* (and hence is not outdated). This means that every snapshot already corresponds to a historical dataset.

[5]Please write an email to dbis-research@informatik.uni-hamburg.de.
[6]https://www.ncsbe.gov/

3 TEST DATA DESIDERATA

Before we describe in which way we used the history of the NC voter register to generate test data in Section 4, we will take a closer look on the desired properties of such a test dataset. A suitable test dataset has to ensure a high

- *quality*, i.e., the test data should enable meaningful evaluation results,
- *usability*, i.e., the user should be able to customize the test data according to her requirements, and
- *reproducibility*, i.e., the user should be able to reproduce the results of past evaluations that used previous versions of this dataset in order to achieve adequate comparability.

In the rest of this section, we will discuss these requirements and the problems that are related with them in more detail. The way we handled them in our generation process will be described in the remaining course of this paper.

3.1 Quality

A test dataset is of good quality if its gold standard is sound, its data contain real(istic) errors of different types and it contains only few exact duplicates.

3.1.1 Soundness of the Gold Standard. A test dataset for duplicate detection consists of a set of data records and the corresponding gold standard that specifies the duplicate status between the individual records. While errors in the actual data are quite desirable (see Section 3.1.2), it is extremely important that the gold standard is sound, because even a small number of incorrectly labeled record pairs (i.e., *false positives* and/or *false negatives*) can render evaluation results completely useless.

In a perfect world, the mappings between the voter records and actual voters are sound. However, almost no dataset is free of errors. Thus, it make sense to perform a soundness check on the test data because, as we illustrate in Figure 3, there may be clusters whose records do not seem to represent the same voter although they share the same NCID. Marking those clusters allows the user to remove or repair them before using the test dataset. Since we often cannot distinguish between sound and unsound clusters with absolute confidence, it does not seem wise to use a Boolean flag as a marker, but to compute similarities which reflect a kind of likelihood that these clusters are sound (i.e., all their records represent the same voter). The user can then use these similarities to decide which risk she wants to take to include unsound clusters into her test data. We refer to this similarity as *plausibility* in the rest of this paper and discuss a calculation of plausibility scores for the NC test dataset in Section 6.2.

3.1.2 Error Diversity. The results of an evaluation with a test dataset are only representative if this dataset contains real-life (or at least realistic) data values and errors. In our case, both are real because they originate from a real-life dataset. Moreover, users want to evaluate algorithms that should later be applied to error-prone data. This requires that the test dataset contains errors of various kinds and not only outdated values. This includes typos, abbreviations, invalid values, inconsistencies and different forms of representation. In other words a test dataset of high quality has to contain several problems of data quality.

3.1.3 Amount of Exact Duplicates. Another aspect that affects the usefulness of evaluation results is the number of exact duplicates contained in the test data. The detection of such duplicates is rather simple and every duplicate detection algorithm – no matter how primitive – should be able to detect them. Thus if this number dominates the number of fuzzy duplicates by far, an accurate detection of the latter becomes less relevant in order to achieve a good evaluation result. For example, if 90% of all duplicate pairs are exact, even the most primitive algorithm would achieve a recall of 0.9 or higher if it is able to compare values on equivalence. Moreover, an algorithm that classifies only the exact duplicates as such and all other pairs as non-duplicates (precision is 1.0) would even achieve a F_1-score of 0.9 which is a pretty good result. However those algorithms are completely useless when it comes to real-life use cases where fuzzy duplicates need to be detected. While this aspect is of little relevance in many approaches to test data generation (the number of exact duplicates is usually very small there), it is of great importance when using historical data, since many of the given snapshots overlap to a large extent, so that their combination leads to many exact duplicates. As we will see in Section 4, by simply combining the individual snapshots of the NC voter register we produced a relative amount of exact duplicates of over 90%.

The actual definition of an exact duplicate pair is that both records share the exact same value in every attribute. However, in the case of the NC voter data, solely removing those duplicates which are completely identical does not solve this problem, because often many of the remaining duplicate records only differ in some minor aspects, such as:

- *Meta Data Attributes*: Many duplicate records only differ in some date values, such as snapshot or registration date, that are less relevant for the duplicate detection process.
- *Time-related Attributes*: The voters' age values increase by one every year and thus cause that some duplicates are no longer exact, although none of the other characteristics of the corresponding person changed.
- *Whitespaces*: Many values contain leading and trailing whitespaces that are simply to detect and remove by trimming all data values in an initial preparation step.

We describe how we addressed this problem in Section 4.

3.2 Usability

Since we aim to provide test data for duplicate detection at scale, the resulting dataset should contain several million records. Besides size, the requirements of the individual users can vary from one evaluation to another. Therefore it is advantageous if the test data can be adapted to the needs of the respective use case in terms of several data characteristics, such as the number of clusters, the cluster sizes, or the degree of heterogeneity (a.k.a. *dirtiness*). This can be accomplished by applying a postprocessing step, which selects a subset of all data carefully. Further options for customization are the removal and merge of attributes, changing the character of the attributes' values. A flexible and unconstrained customization requires that the test dataset contains (i) many duplicate clusters of various sizes and (ii) duplicate records of different degrees of heterogeneity, so that the user has a large set to choose from. In addition, it requires that the user can adjust the characteristics of the test data with relative ease. This can be supported by storing all records of one cluster together and providing precalculated heterogeneity scores.

The heterogeneity of the individual duplicate clusters (or duplicate pairs) represents the degree to which the duplicate records differ from each other. At the same time, it can be considered as a measure on the difficulty of detecting the fuzzy duplicates within this dataset because duplicates are usually the more difficult to detect, the more their values differ. Thus, this information does

Table 2: Statistical results of the generation process

duplicate removal	*#records*	*#dupl. pairs*	*cluster size*		*#removed*	
			avg.	*max.*	*records*	*pairs*
no	522.5 M	12,108.2 M	38.50	399	0%	0%
exact	166.3 M	1,225.0 M	12.26	104	68.2%	89.9%
trimming	120.8 M	648.2 M	8.90	77	76.9%	94.6%
person data	58.7 M	136.7 M	4.33	51	88.8%	98.9%

*The number of objects (i.e., clusters) was always 13.57 M.

not only provide interesting insights into the nature of the test data, but also allows the user to customize the level of difficulty of her test dataset individually by filtering out clusters/records whose heterogeneity is not within a requested range. This can be useful when the user wants to test her algorithms with datasets of different degrees of dirtiness. It is important to note that such filtering can theoretically be performed on any test dataset. However, only a large number of clusters and records allows the user to compose arbitrary subsets without running into the problem of producing a too small output.

3.3 Reproducibility

The NC voter register is subject to constant change and new snapshots are published regularly. This gives the opportunity to extend the generated test dataset on a regular basis, too, which does not only provide data on new voters (i.e., more duplicate clusters), but also new data on already existing voters (i.e., larger duplicate clusters and higher degrees of duplicate heterogeneity). In general, the longer the time span covered by a test dataset, the more outdated values it contains. Moreover, a longer time span increases the chance of obtaining outdated values even for attributes that do not change very frequently (e.g., the last name).

In performance and quality evaluation, *reproducibility* [27] (a.k.a. *repeatability* [28]) is an important aspect because it is necessary to enable a fair comparison between (the evaluation results of) different algorithms especially if they are evaluated at different times by different parties. In this context, reproducibility means that another evaluation process (or at least its experimental setting) can be reproduced exactly. This includes the use of the same test data. Thus, test datasets that change over time pose a problem for reproducibility, especially if their size does not allow a separate storage of every intermediate version. To solve this problem, the datasets must be enriched by information that allows the user to reconstruct any of the old versions on request. It is important to note that such a reconstruction does not only apply to the actual records, but also all meta data (e.g., statistics and similarities) that are stored to describe them. We discuss how we ensure reproducibility for our test dataset in Section 5.1.

4 TEST DATA GENERATION

Since every snapshot of the voter register already contains outdated records and we wanted to reduce execution time as well as the number of exact duplicates as much as possible, we started to experiment with a single snapshot. Because we planned to use as much data as possible and the size of the snapshots grew monotonically over time, we used the most recent one. At the time of our analysis, this was the one created at 2021-01-01 and indeed, among all available snapshots, it contains the most records having the voter status *removed* (and thus are potentially outdated). The total number of records in this snapshot is 15,863,484 (8,308,925 *removed*) which corresponds to 3.04% of all voter records. However, the experiments showed that the amount of historical information stored in the individual snapshots is rather low compared to the whole history and thus only provides small clusters (see Figure 1a). Therefore, we also evaluated the entire voter history containing all snapshots available. By doing so, we examined a total of 522,463,029 records.

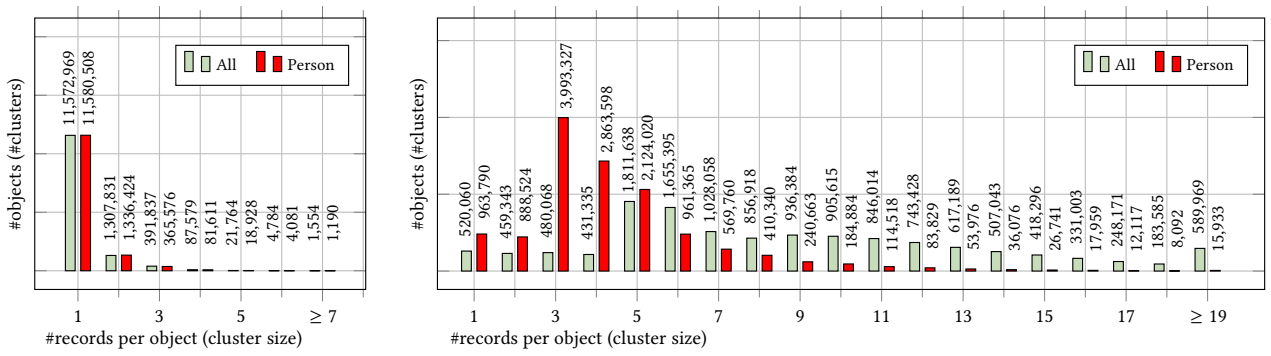
One of our goals was to analyze the amount of (near) exact duplicates within the resulting test data. Moreover, we think that most potential users are only interested in the personal data of the voters, since the election and district attributes are very case-specific. Therefore, we executed our approach four times: (i) one time without removing any duplicates, (ii) one time with removing all exact duplicates, (iii) one time with removing all duplicates that were exact after their values have been trimmed (i.e., leading and trailing whitespaces were deleted), and (iv) one time with removing all duplicates whose personal data were equivalent (after trimming attribute values). To check the equivalence of duplicate records efficiently, we used the *Message-Digest Algorithm 5* (short MD5) to calculate a hash value for each record. A record was then not imported into the test dataset when it already contained a record with the same hash value. Of course, collisions between non-exact records cannot be excluded for sure, but such a collision only means that the test dataset loses a duplicate record and thus does not have severe consequences if it happens a few times[7]. The input to the hash function is the concatenation of the values of all relevant attributes to a single large string. As mentioned in Section 3.1.3, some meta data and time-related attributes can reduce the number of near exact duplicates drastically and therefore were not included into the concatenation. These attributes are the different dates (snapshot, load, registration and cancellation) and the age[8].

The number of resulting objects (i.e., duplicate clusters), records, duplicate pairs, the average and maximal number of records per object (i.e., duplicate cluster size), as well as the number of removed records and duplicate pairs are listed in Table 2. The number of distinct duplicate clusters (i.e., objects) per cluster size (i.e., number of records per object) is presented in Figure 1b (one time for all attributes and one time for the personal data only). Using a single snapshot did not produce any exact duplicate which was even less than expected. In contrast, using all snapshots produced hundred of millions of exact duplicates. Obviously, the average number of records per voter decreased when these exact duplicates were removed (e.g., 8.90 without whitespaces) and decreased further on when we restricted the data to the personal attributes (4.33), but was still large compared to the single-snapshot approach (1.18). In total, the number of records that were removed because of being exact duplicates was up to 76.9% and the number of removed duplicate pairs was up to 94.6% when all attributes where taken into account and up to 88.8% and 98.9% if only person data was considered. These large amounts of exact duplicates illustrate the importance of their removal, because otherwise every evaluation of duplicate detection algorithms using these data would suffer from the effects described in Section 3.1.3.

To estimate the value of future snapshots, we counted the number of new clusters (i.e., the snapshot contains an NCID that

[7] MD5 produces 128 bit hashes, which means that it is relatively unlikely that the hash values of two different inputs collide.

[8] In the case of age values, the most obvious solution is to transform them into years of birth because the latter do not change. However, such a calculation would enclose a part of the dates of birth, which were originally removed from the data due to privacy reasons. We have therefore decided to use them only for internal calculations (e.g., plausibility) and not to store them in the resulting test data.



(a) Single Snapshot **(b) All Snapshots**

Figure 1: Distribution of the number of records per object (i.e., cluster size) after removing exact duplicates (trimming)

was never used before) and new records (i.e., the snapshot contains a record that was not part of any of the previous snapshots) per snapshot-year. These numbers are presented in Table 1 (note that the second includes the first). The last two columns show the percentage of rows that result in a new record (*new record rate*) and the percentage of new records that lead to a new cluster (*new object rate*). As expected, the number of new clusters and new records is the highest for the first snapshot. Surprisingly, the numbers of the following snapshots vary enormously (we expected an almost constant number). Investigations revealed that in some snapshots the formats of one or two attributes changed (e.g., from ‚64TH HOUSE' to ‚NC HOUSE DISTRICT 64') so that each of their records were considered to be ‚new' even if – apart from that – they were identical with one of the already existing ones. This again shows us the importance of providing the user with an instrument that allows her to filter out records based on their similarities (see Section 6.5). However, as one can see, even the last five snapshots contributed a significant amount of new clusters and records to the test data. Thus, we can expect the same for future snapshots.

5 TEST DATA STORAGE

The voter history is originally given as a set of TSV files. However, we want to store our test dataset by using a data model that is more suitable with respect to its later usage, i.e. to evaluate duplicate detection algorithms as well as potential extensions with data from new snapshots. As a consequence, the data model has to satisfy three essential requirements:

- To customize the test data, we need to select and reduce duplicate clusters based on user-defined specifications. Moreover, when we integrate additional snapshot data, we need to calculate statistics by comparing duplicate records (e.g., plausibility and heterogeneity). Both require fast and collective access to all records of the same cluster.
- Every record of the voter data has 90 attributes, but only a few records have values for district-related attributes. This means that millions of records have missing values in at least 38 attributes. Thus, we require the underlying data model to provide an efficient handling of sparse data.
- Working on hundreds of gigabytes requires scalable software solutions.

Schemaless NoSQL data models are much better suited to store sparse data than the relational model which requires the definition of a rigid schema. Moreover, many NoSQL data stores are designed to handle *aggregates* each of which is a collection of related data that we wish to treat as a unit [29]. Thus, they allow an easy and efficient way to access all the records of a certain person as we need it for customization and future extensions. Among all the available NoSQL data stores, we decided to use the document store MongoDB [22]. In contrast to the relational data model which is *aggregate-ignorant*, document stores are strongly *aggregate-oriented* [29] because they allow to (i) group records by storing them within the same document and (ii) nest different documents hierarchically. Furthermore, MongoDB is highly scalable. Besides its schemaless structure, MongoDB has three features that are especially helpful for this work [22]:

- *Indexes:* Since our test dataset contains millions of nested documents, indexes are very important to efficiently select those documents from the dataset.
- *Aggregation Pipeline:* Multi-stage pipelines can be used to transform documents into an aggregated result. Available pipeline stages provide tools for filtering, transformation, grouping and sorting. These pipelines enable users to extract relevant subsets of the data and thus to customize their own test datasets.
- *Compass:* MongoDB has a powerful GUI called *Compass*. It enables the user to easily interact with the stored data with full CRUD functionality. It is very helpful for exploring, generating, adjusting and using the test data. Moreover, it allows to monitor load jobs of new snapshots and helps to identify mistakes at an early stage.

In our case, we created one document for every person (i.e., duplicate cluster) that in turn contains a document for every record of this person (which are grouped into an array) and in addition a document containing some relevant meta data including the hash values of the stored records. Since most users will be interested in the personal data only, we split every record into four parts (person, district, election and meta) and stored them into different subdocuments.

5.1 Future Updates & Reproducibility

The NC State Board of Elections publishes a new snapshot at every election and every New Year's Day. Moreover, we have observed that they published some old snapshots belatedly (e.g., the snapshot from 2010-11-02 was first published in May 2019). To improve our test dataset both in size and heterogeneity, we will update it regularly.

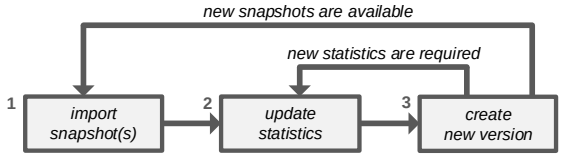


Figure 2: Update process for new snapshots and statistics

5.1.1 Update Process. The update process is depicted in Figure 2 and consists of three steps. The first step corresponds to a (parallel or sequential) import of one or more new snapshots. In the second step, current statistics are updated and (if required) new statistics are calculated. In the last step, a new version number is associated to the test data, versioning-related meta data are updated and the new version is published. As illustrated by this figure, the update process (and thus the creation of a new version) can be triggered by two reasons:

- new snapshots are available, or
- new statistics are required.

Logically, in the second case, the first step is skipped and the update process starts immediately with the second one.

5.1.2 Reproducibility. As described in Section 3.3, the import of new data poses some challenges to the repeatability of evaluation processes that were using previous versions of our test dataset. Since no record is ever removed from the test dataset, it grows monotonically, which means that the set of all records of the current version will always be a subset of all records of every future version. Thus, theoretically, it is sufficient to add a field to every record which is monotonically increasing with every new update. This field can be the date of import or the number of the first version containing this record (the snapshot date is not suitable because it is not monotonically increasing due to belatedly published snapshots). To reconstruct an earlier version, the user can use this field to filter out all records whose field value is greater than the value of the desired version.

However, we also want to allow users to limit their evaluation to an arbitrary subset of snapshots (e.g., a certain time interval). For doing this, we additionally store an array with the dates of all snapshots containing the corresponding record.

To reconstruct statistics, such as the number of records or snapshots per cluster, we enriched the meta data of every duplicate cluster by a map that counts how many new records were inserted per snapshot. The reconstruction of similarity scores is discussed in Section 5.2.

5.2 Storing Similarity Scores

To support users in customizing their data by (i) removing further near exact duplicates, (ii) repairing potentially unsound clusters, and (iii) restricting the data to a certain range of heterogeneity, we associate every record with three version-similarity maps (one for plausibility and two for heterogeneity). Every value of each of these maps corresponds to another map that assigns a similarity score to each of the previously existing records of the same cluster. Since the order of these records never change, this approach does not only avoid expensive recalculations, it also enables reproducibility because existing similarity scores are never updated or deleted. While the first heterogeneity map takes all attributes into account, the second is limited to the personal attributes in order to facilitate a customization of personal data[9]. In addition every cluster is associated with three version-similarity maps storing the aggregated values of their records.

Our understanding of plausibility and heterogeneity may change over time and/or we just may want to use other similarity measures to compute them. The versioning of the similarity scores protects reproducibility against such future changes, because we create a new version every time we use a new measure.

[9]Note, the plausibility is already limited to personal attributes (see Section 6.2) and thus does not need to be stored twice.

	NCID†	*last_name*	*first_name*	*midl_name*	*sex*	*age*	*year*◇
r_1	XX001	LARRELL	LEWIS	ANTWAN	MALE	17	2014
r_2	XX001	LEWIS	LARRELL	ANTAWN	MALE	18	2015
r_3	XX001	LEWIS	LARRELL	A	MALE	22	2018
r_4	ZZ002	FIELDS	MARY	ELIZABETH	FEMALE	62	2012
r_5	ZZ002	BETHEA	JOSHUA	ELIZABETH	MALE	92	2014

†The NCIDs are pseudonymized for data privacy reasons.
◇The snapshot year in which this record was collected.

Figure 3: Examples of erroneous and unsound clusters

6 EXPERIMENTAL EVALUATION

When we explored the snapshots, we noticed several errors within the data. Some records contain typos, abbreviations, or have values confused between attributes. Moreover, some notations have changed over time (e.g., ‚1ST CONGRESSIONAL‘ vs. ‚CO. DISTRICT 1‘ or ‚66 AND ABOVE‘ vs. ‚Age Over 66‘). One example is presented in Figure 3. The first and last names of at least one record of voter XX001 got mixed up. In addition, either the middle name of r_1 or r_2 contains a typo (‚ANTWAN‘ vs. ‚ANTAWN‘) and the middle name of r_3 is abbreviated. Remember that a proper evaluation of duplicate detection algorithms requires errors of many different types and not just outdated values. Thus, such real-life data errors are very welcome in our test dataset, since they challenge the detection of duplicates, but do not corrupt the gold standard. However, we also detected some duplicate clusters that contain records that hardly refer to the same person. An excerpt of one of those examples is depicted in Figure 3 where the two records r_4 and r_5 share the same NCID, but obviously describe different persons. Such unsound clusters are a real threat to the quality of our test data because they spoil the gold standard and thus will negatively affect every future evaluation if they remain in the dataset.

In order to evaluate the quality and usability of our test dataset beyond these first impressions, we conducted a series of experiments, which are described in the rest of this section.

6.1 Evaluated Datasets

To better understand the results of the evaluation of our test dataset, we compare them with those of three manually labeled test datasets that are commonly used in the literature. We acquired all three datasets as TSV files[10] from the dataset repository of the Hasso Plattner Institute[11].

- *Cora*: This dataset contains bibliographical information on scientific papers including title, authors, publisher and year. The schema of the TSV file consists of 17 attributes including an artificial id. The file contains 1,878 records which form 182 clusters.
- *Census*: This dataset contains personal information including name values (first, middle and last), an address and a zip code per person (6 attributes in total). It contains 841 records which form 483 clusters.

[10]We used the non-prepared versions where special characters are not removed.
[11]http://hpi.de/naumann/projects/repeatability/datasets

Table 3: Characteristics of evaluated datasets

	Cora	Census	CDDB	NC1	NC2	NC3
#records	1,879	841	9,763	24,761	22,739	25,530
#attributes	17	6	7	38	38	38
#duplicate pairs	64,578	376	300	19,916	15,993	22,735
#clusters	182	483	9,508	10,000	10,000	10,000
#non-singletons	118	345	221	10,000	10,000	10,000
max. clustersize	238	4	6	7	7	8
avg. clustersize	10.32	1.74	1.03	2.45	2.27	2.55
max. heterog.†	0.63	0.46	0.65	0.25	0.43	0.72
avg. heterog.†	0.171	0.15	0.217	0.106	0.305	0.433

† The presented heterogeneity scores are pair-based.

- *CDDB*: This dataset includes information on 9,763 music CDs randomly extracted from freeDB[12]. In the TSV version of this dataset, all tracks of a CD are concatenated to a single string by using the pipe symbol as a delimiter. After doing so, the schema of this file consists of 7 attributes. The 9,763 records form 9,508 clusters.

For all three datasets the duplicate information is provided by a list of pairs. Several characteristics of these datasets are presented in Table 3. Interestingly, the duplicate distributions of these sets are quite different. Whereas the Cora dataset contains very large clusters (up to 238 records) and its average cluster size is 10.32, the maximal and average cluster sizes of the Census and CDDB datasets are quite small (1.74 or 1.03 respectively). The three datasets NC1 to NC3 are described in Section 6.5.

6.2 Plausibility Check

As we have explained in Section 3.1.1, it is very important that the gold standard of the test data is sound. However, as shown in Figure 3, we have also seen that this does not always seem to be the case. To keep the threat of an unsound gold standard to a minimum, we performed a quality check by calculating a plausibility score for every pair of duplicate records.

In this plausibility check, we have the basic assumption that all records of the same cluster are duplicates and the similarity scores should only reflect (significant) contradictions to this assumption. Consequently, the similarity measure should compensate simple errors and differences in data representation as we know it from duplicate detection algorithms. Due to our basic assumption, this compensation can be even stricter. Therefore, word confusions within an single attribute value or between different values as well as missing or abbreviated values should not reduce similarity at all, because they are more an indication of unknown or erroneous values than a clear sign of a non-duplicate. Moreover, to compensate outdated values we should only use attributes whose values rarely change and that are either very identifying (i.e., two records with the same value are likely duplicates) or discriminating (i.e., two records with different values are likely no duplicates). In our use case, we decided to use:

- the three name values (first, middle and last),
- the sex code,
- the year of birth (which we derived from the snapshot-date and the age value), and
- the place of birth.

It is not uncommon that values are confused between the three name attributes. Thus, we computed a single *name similarity* before combining it with the similarity scores of the other attributes.

[12] http://www.freedb.org/

To compensate errors in the name order, but also within the individual name values (e.g., typos), we computed the name similarity by using the hybrid Generalized Jaccard Measure [8] with an extended version of the Damerau-Levenshtein Similarity [4] as the internal token similarity measure, i.e.:

$$sim_{name}(o_i, o_j) = GenJacc_{DamLev}(names(o_i), names(o_j)) \quad (1)$$

where $names(o_i) = \{fname(o_i), mname(o_i), lname(o_i)\}$.

The Damerau-Levenshtein Similarity was extended to a proper handling of missing and abbreviated values. The comparison to a missing value results in a similarity of 1. The same holds true if one token is a prefix of the other because in both cases we do not have any evidence to mistrust the given data. In the case of the sex code, typos and different representations can almost be excluded for sure. Thus there are actually only four possibilities: The compared values agree (i.e., $sim_{sex} = 1$), disagree (i.e., $sim_{sex} = 0$), one of them is undesignated (i.e., it has the value ‚U') or missing. Since we do not have any contradiction in the later two cases, we set the similarity to $sim_{sex} = 1$, too.

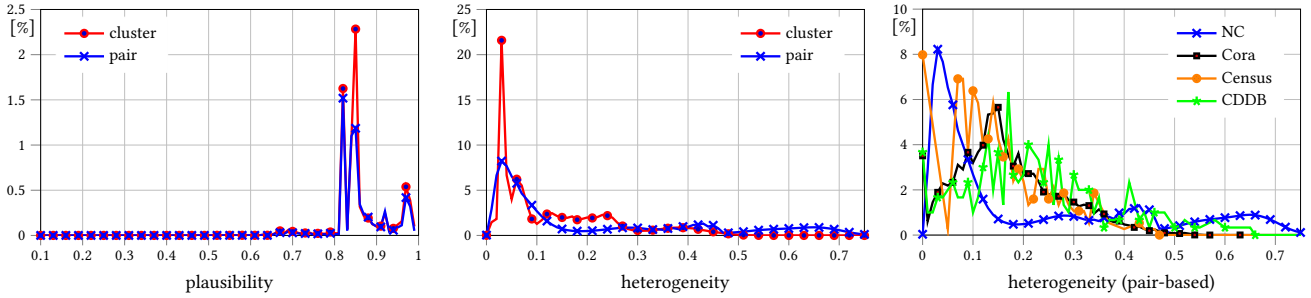
We computed the year of birth (short YoB) as *snapshot-date* − *age*. Since the actual YoB can be one year earlier if the person has not yet had birthday when the snapshot was made, we introduced a tolerance of 1. Moreover, we assumed a similarity of 0 if the age difference was 10 or greater. This led to the following formula:

$$sim_{YoB}(o_i, o_j) = 1 - min\left(1, \frac{max(0, |YoB(o_i) - YoB(o_j)| - 1)}{10}\right) \quad (2)$$

In the case of the place of birth we simply computed the extended Damerau-Levenshtein Similarity between the two values. The final similarity score was then calculated as the weighted average of the previously presented scores where we considered the name similarity to be more important (weight 0.55) than the others (each 0.15). A cluster is already unsound, if only one of its records refers to another voter regardless of how plausible the other records are actually duplicates. Thus, we computed the plausibility of a cluster as the minimal plausibility of its records.

We performed our plausibility check on the dataset with 120 million records (exact duplicates were removed after trimming). The results show that only a few clusters of this dataset are highly suspicious to be unsound. The average cluster plausibility is 0.988. 91.7% of all clusters (and 93.3% of all duplicate pairs) have the maximum possible value 1.0. The distribution of the remaining 8.3% (or 6.7% resp.) are presented in Figure 4a. The minimal plausibility of all clusters (and pairs) is 0.06. 6.4% of all clusters have a plausibility lower than 0.9, 0.47% (=61,548 clusters) lower than 0.8 and only 0.0049% (=641 clusters) lower than 0.5. The pair-based values are similar. As a comparison, the two clusters from Figure 3 have a plausibility of 0.82 (XX001) and 0.33 (ZZ002) respectively which matches our intuition that the differences in the first cluster are probably the result of data errors in the name values while the second cluster contains obvious non-duplicates.

An appropriate scoring of plausibility heavily depends on the domain of the data, since we should only use attributes that are less volatile and are either very identifying or discriminating. Moreover, it also depends on the quality of the data, since typical error patterns (e.g., an incorrect encoding of special symbols) are no significant evidence for an unsound cluster and should therefore be compensated in the scoring process. It is therefore difficult to make comparisons between the plausibility of datasets defined on different schemas without creating any noticeable bias. For this reason, we decided not to include such a plausibility calculation for the Cora, Census and CDDB datasets.



(a) Plausibility of NC test dataset **(b) Heterogeneity of NC test dataset** **(c) Heterogeneity of NC, Cora, Census and CDDB**

Figure 4: Plausibility and heterogeneity distributions of clusters and duplicate pairs for different datasets

6.3 Duplicate Heterogeneity

To score the heterogeneity between two duplicate records we want to take all their differences into account and thus do not actually want to apply measures that compensate them. At the same time, there are differences that should cause a larger heterogeneity than others. For example, difference in upper and lower case or confusions of tokens are less significant than replacing the original strings with completely different letters or tokens.

To address the problem resulting from uppercase letters, we decided to compare every two values one time with and one time without lowercasing them. To address the problem of token confusions, we decided to compare every two values one time with a sequential and one time with a hybrid similarity measure. Together this results in four comparisons for every two values. We used the average of the four resulting scores as final similarity. As the sequential similarity measure, we chose Damerau-Levenshtein. Since the Generalized Jaccard Measure is computationally too expensive when working on 90 attributes, we used the Monge-Elkan Similarity[13] [23] as the hybrid measure instead (using Damerau-Levenshtein as the internal token similarity measure)[14]. To calculate the heterogeneity between two records, we used the weighted average of their inverse value similarities (i.e., the records are the more heterogeneous, the less similar they are).

If the resulting heterogeneity scores should reflect any kind of difficulty to detect the corresponding fuzzy duplicates, identifying attributes, such as names, should be weighted higher than others. At the same time and in contrast to plausibility, we want to compare the heterogeneity (i.e., dirtiness) of different datasets in order to enable an evaluation of algorithms with respect to a varying dirtiness of the data. To ensure a fair comparison, no context knowledge and external expertise should be included into the scoring process and all necessary information should be taken from the dataset itself. To accomplish this, we used the same similarity measure for all attributes and weighted every attribute by its uniqueness[15], where we quantified this uniqueness by the attribute's entropy [19]. Since duplicate records distort these uniqueness scores (e.g., an otherwise unique id can occur multiple times), we initially created a canonical record[16] per duplicate cluster and used them to compute the weights. The heterogeneity of a cluster was then computed as the average heterogeneity of its records. Since a consideration of clusters of size one is pointless, we restricted our analyses to clusters with at least two records.

The results of our analysis are depicted in Figure 4b. They show that – despite of outdated values and other data errors – most of the duplicate records are very similar and the dataset as a whole is quite clean and homogeneous. However, since we removed exact duplicates, almost none of the clusters is completely homogeneous and most of them have a heterogeneity of around 0.03 (21.6%). In the case of the duplicate pairs, we do not have such a large peak, but also here you can see, that most values are in the range between 0.02 and 0.06. Considering this fact, the average heterogeneity (0.13 for clusters and 0.218 for pairs) is surprisingly large. The maximal heterogeneity (0.79 for clusters and 0.88 for pairs) is also very high. As a comparison, the two clusters from Figure 3 have a heterogeneity of 0.395 (XX001) and 0.366 (ZZ002) respectively. Interestingly, the less plausible cluster is more homogenous. That is because although cluster ZZ002 contains records referring to different persons, these records form two very homogenous groups (one with six records similar to r_4 and one with four records similar to r_5).

To score the heterogeneity of the Cora, Census and CDDB datasets, we used the same settings (i.e., the same similarity measures and all attributes are weighted based on their entropy). The pair-based distributions of all three datasets are depicted in Figure 4c. The heterogeneity scores of the Cora dataset are almost normally distributed. Most of them have a value of 0.15, the maximal value is 0.63 and the average is 0.171. The Census dataset has three peaks at 0, 0.07 and 0.1, a maximal heterogeneity of 0.46 and an average of around 0.15. In general, except of the large number of very homogenous pairs, its distribution has some similarity to this of the Cora dataset, but is less regular. The CDDB dataset is the dirtiest of them. Its maximal heterogeneity is 0.65 and its average is 0.217. The high heterogeneity results primarily from the fact that many language-specific symbols, such as accents, were incorrectly converted when the dataset was created. Thus, many duplicate records are very dissimilar if we do not compensate those irregularities in the matching process or repair it during preparation.

If we compare these distributions with the one from the NC dataset, we see that there is only little resemblance to our voter data although the average of the NC dataset is very close to that

[13] Since this measure is non-symmetric, we computed it in both directions and used the average as final score.

[14] There are millions of possible similarity measures (including various settings) and we had to choose one even though this could possibly generate a bias in the evaluation processes using this test data. However, as illustrated in Section 6.5, this bias was almost not existing in our experiments.

[15] To ensure reproducibility, these weights must not change although the dataset will grow with future updates. We therefore limited their computation to the first 5 million clusters and then hard-coded them into the source code.

[16] These records are built by using the most frequent value per attribute.

Table 4: Statistics of different irregularities within the NC, the Cora and the Census datasets

			NC (total 137 M pairs)			*Cora* (total 65 K pairs)			*Census* (total 376 pairs)		
			most frequent	*frequency*		*most frequent*	*frequency*		*most frequent*	*frequency*	
	error type	*example*[†]	*attribute*	*total*	*in %*	*attribute*	*total*	*in %*	*attribute*	*total*	*in %*
singleton[◇]	outlier	age = 5091	age	280 K	0.48	year	20	1.06	last_name	5	0.59
	abbreviation	midl_name = A.	midl_name	7.4 M	12.6	booktitle	1	0.05	middle_name	671	79.8
	missing	mail_addr3 = *null*	mail_addr3	58.7 M	100	institution	1.7 K	86.8	middle_name	170	20.2
pair-based[★]	typo	ADELL ↔ ADELE	midl_name	1.2 M	0.87	title	15 K	22.6	last_name	243	64.6
	OCR-error	DICOL3 ↔ DICOLE	last_name	1.3 K	0.00	-	-	-	-	-	-
	phonetic	WHITE ↔ WYATT	midl_name	1 M	0.75	title	29 K	44.4	last_name	200	53.2
	prefix	KIM ↔ KIMBERLY	midl_name	5.4 M	3.94	volume	19 K	29.1	first_name	65	17.3
	postfix	BRAGG ↔ FORT BRAGG	last_name	230 K	0.17	pages	5 K	8.03	street_address	5	1.33
	formatting	JRS RIDGE ↔ J.R.S RIDGE	midl_name	208 K	0.15	title	27 K	42.4	street_address	20	5.32
	transp. tokens	KIM DUC ↔ DUC KIM	race_desc	748 K	0.55	authors	202	0.31	-	-	-
	confused values	(LUKE, HAL) ↔ (HAL, LUKE)	first/midl_name	21 K	0.02	title/booktitle	20	0.03	-	-	-
	integrated value	(SUE ANN, *null*) ↔ (SUE, ANN)	midl/last_name	244 K	0.18	title/year	4	0.01	-	-	-
	scattered values	(NGAN HA, THI) ↔ (NGAN, HA THI)	midl/last_name	24 K	0.02	-	-	-	-	-	-

[†]Selected from any attribute of the NC dataset.
[◇]Singletons are normalized using the total number of records (NC = 58.7 M, Cora = 1,879, Census = 841).
[★]Pair-based irregularities are normalized using the total number of duplicate pairs (NC = 136.7 M, Cora = 65 K, Census = 376).

of the CDDB dataset. Interestingly, the majority of pairs of the NC dataset is much cleaner than those of the other three datasets, but it has also a higher percentage in the range between 0.4 and 0.7. Thus, our dataset contains many homogenous records, which may need some additional pollution (see Section 8). However, as we will show in Section 6.5, its large dispersion allows us to easily achieve an average heterogeneity of 0.433 (which is significantly higher than this of the CDDB dataset) by adjusting the voter data based on the precalculated heterogeneity scores.

6.4 Diversity of Error Types

To allow an extensive evaluation of the capabilities of duplicate detection algorithms, we chose source data whose collection process promises many different types of errors. To test this assumption, we conducted a statistical analysis on the personal attributes of our test dataset by searching for several kinds of irregularities within these data (see Table 4). Here, we distinguished between irregularities that can be identified by analyzing single records (so-called *singletons*) and ones that can only be detected by comparing two duplicate records (so-called *pair-based* irregularities). In the first case, we evaluated every record individually leading to a frequency that can be normalized using the total number of records. In the second case, we compared every two duplicate records, counted the number of times the individual irregularities occur and normalized them using the total number of duplicate pairs. Moreover, we distinguished between irregularities that concern a single attribute and those that concern multiple attributes (record-level). We evaluated the following singletons:

- *outlier*: A value that is outside a predefined range (e.g., age > 110) or contains a character that is unusual for its associated domain (e.g., the first name ‚X ÆA-12‘)[17].
- *abbreviation*: A value that consists of a single letter, possibly followed by a punctuation mark.
- *missing*: A value that is null, an empty string or any other value indicating missing information (e.g., ‚-‘ or ‚unknown‘).

As pair-based irregularities, we analyzed:

- *typo*: Two values whose lowercase versions differ only in one character or contain a character transposition. These are exactly those values having a Damerau-Levenshtein distance of 1. In order not to interpret a complete replacement of one value by the other as a typing error, we only considered values longer than two characters.
- *OCR-error*: Two distinct string values which only differ at those positions where one of them has a digit. If both characters are digits, they need to be identical.
- *phonetic error*: Two values that are not identical after removing non-letter characters, are both longer than two and have the same Soundex code.
- *prefix/postfix*: Two values where one of them is a prefix/-postfix of the other after removing a potential punctuation mark from the end of the shorter value. Such pre- and postfix situations indicate abbreviations or forgotten token/characters.
- *different formatting*: Two values that only differ in non-alphanumerical characters (e.g., a hyphen, space or punctuation mark between tokens).
- *transposed tokens*: Two values whose token sets are identical, but their token order is different.
- *confused values*: Two records whose values are confused between two different attributes (e.g., the first and last name of one record are transposed).
- *integrated value*: Two records where in one of them the value of one attribute is integrated into another (e.g., a middle name stored as a second token in the first name).
- *scattered values*: Two records having the same set of tokens assigned differently to two attributes. To avoid possible overlaps with the previous two types, we only counted scattered values that are not integrated or confused.

Obviously some of these irregularities overlap (or sometimes even include each other) so that we counted some errors for more than one type. For example, some OCR-errors are also typos. Moreover, it is important to note that we consider these irregularities as indications of particular error types, but cannot always classify them with absolute confidence. For instance, not every two distinct values that have the same Soundex code are an actual phonetic error. This also applies to irregularities on record-level. Not every assignment of the same value to different

[17]Note that not every outlier corresponds to an actual data error.

attributes corresponds to a mistake. For example, in the U.S., it is not atypical to take the old last name as the middle name when getting married. Such a constellation can therefore also indicate an outdated record instead of a data entry error. Nevertheless, the errors are real even if their assignment to the individual types may be disputed. It should also be mentioned that our definitions of the individual error types do not cover them completely so that our analysis was not able to find every actual error. For example, OCR errors that do not contain digits (e.g., ‚Tim‘ vs. ‚Tirn‘) were not recognized as such. However, despite these minor inaccuracies, we think that our experiment provides a good overview of the wide variety of different error types contained in the voter data.

The results of this analysis are presented in Table 4 (grouped by singletons and pair-based irregularities), were we list the absolute values in combination with their percentages. To achieve greater comparability, we also evaluated the Census and the Cora datasets. We selected the Census dataset because it has a similar domain as our voter data and selected the Cora dataset because it has similarly large clusters. As we can see, the percentages of the Census and the Cora datasets are much higher than those of the NC dataset. For example, the percentage of typos in the attribute *last_name* of the Census dataset is 65%. This means that out of 376 duplicate pairs, 243 differed in this attribute by only one character or had two consecutive characters transposed. Although the percentages of the NC dataset are much smaller than those of the Cora and the Census datasets, the absolute numbers are many times larger. This shows the potential for customizing smaller datasets with higher error percentages, but still containing million of records. Moreover, the NC dataset contains irregularities that are (almost) not contained in the Cora and Census datasets. Examples are OCR-errors or errors that affect more than one attribute.

6.5 Usability

There are various ways to customize a test dataset by using the precalculated heterogeneity scores. In our experiments, we sketched a very simple approach and let the development of more sophisticated approaches to the user (or future research resp.). This approach consists of three steps. In the first step, we defined a lower and an upper bound $h_\perp$ and $h_\top$ for the heterogeneity scores. In the second step, we randomly selected 100,000 clusters from our whole dataset, scanned over all records of every cluster in their sorting order and removed every record whose heterogeneity to its preceding (not removed) records was not in the requested range $[h_\perp, h_\top]$. In the third step, we sorted the reduced clusters by their size and selected the 10,000 largest clusters as test input. We applied this approach for the three settings $(h_\perp, h_\top) \in \{(0.06, 0.2), (0.2, 0.4), (0.3, 0.7)\}$ while restricting the schema to the personal attributes. The result are the three test datasets NC1, NC2 and NC3. The characteristics of these datasets are depicted in Table 3. As these numbers show, even though we only used 0.07% of all clusters as input, these datasets are larger than the Cora, Census and CDDB datasets.

To evaluate the difficulty of detecting fuzzy duplicates within the individual datasets, we applied three duplicate detection algorithms each using another similarity measure[18] (the same for all attributes). The first measure (short *ME/DL*) was the same combination of the Monge-Elkan and the Damerau-Levenshtein Similarity as we used it to calculate the heterogeneity scores (see Section 6.3). The other two measures were the Jaro-Winkler Similarity (sequential) and the Jaccard Similarity using trigrams (token-based) [8]. The similarity of two records was always computed as the weighted average similarity of their values. Since we observed that the name values are sometimes confused between the individual attributes, we matched every combination of them and used the 1:1 matching with the highest similarity for aggregation. To weight the individual attributes we used again their entropy. In this case, however, we calculate it using all records (i.e., including the duplicates), since the user does not know these duplicates in advance. In addition, the entropy was calculated solely based on the records of the customized datasets. Thus, the resulting weights differed from the ones we used to calculate the heterogeneity scores (e.g., 0.66 vs. 0.48 for the first name). In the case of the larger datasets (CDDB and NC1-NC3), we reduced the initial search space by applying a multi pass of the Sorted Neighborhood Method [23] where we conducted one pass for each of the five most unique attributes and used a window of size $w = 20$. Since a few true duplicate pairs were lost through this reduction (always less than 1%), we added them back to the search space before starting the actual matching process.

The results of these duplicate detection algorithms applied to the different test datasets are depicted in Figure 5. As we can see in Figure 5a to 5c, the quality of the duplicate detection algorithms decreased when we increased the heterogeneity of the test data, since the more difficult it was to separate the duplicate from the non-duplicate pairs. In the first case, the test dataset was very clean and we could achieve almost a perfect F_1-score for all three measures. Moreover, for two out of three measures, this score was high for every threshold between 0.65 and 0.85, which made it easier to select an appropriate value for this threshold without knowing these numbers. In the case of the second dataset, the maximal F_1-score was still pretty solid (i.e., close to 0.8), but the threshold had to be set much more carefully and the quality of a setting already depended on the individual measures. For example, for Jaccard the best threshold was 0.57, but for Jaro-Winkler it was 0.75 and there was no threshold that worked well for all measures. Finally, in the case of the last dataset, the maximal F_1-score decreased significantly and even a score of 0.4 was hard to achieve. All this shows that the precalculated heterogeneity scores can be perfectly used to adjust the test data to increase the difficulty of detecting fuzzy duplicates. Moreover, as we can see, the ME/DL Similarity did not perform better than the Jaccard Similarity which shows that using this measure to calculate the test datas' heterogeneity scores has not produced any noticeable bias. Finally, when we compare the results of the three customized test datasets NC1 to NC3 with the results of the Cora, Census and CDDB datasets, we see that they show similar patterns as NC2 in terms of the maximally achieved F_1-score and the shapes and positions of the individual graphs. Only the Census dataset differs a little bit, because here Jaro-Winkler scored much better than for the other datasets and the single graphs correspond less to a bell shape. In summary, this shows that the sheer amount of data of our original test dataset enables us to create test data that are cleaner (NC1), equally clean (NC2) and dirtier (NC3) than these real-life use cases giving us the opportunity to design our test data in the way our evaluation goals require. We repeated this experiment with different parts of our original test dataset as input. Since the compositions of the generated datasets differ slightly, there were also slight differences in the resulting graphs, but the findings were always the same.

[18] Here, we tried to cover a wide range of measures by using a hybrid, a sequential, and a token-based measure.

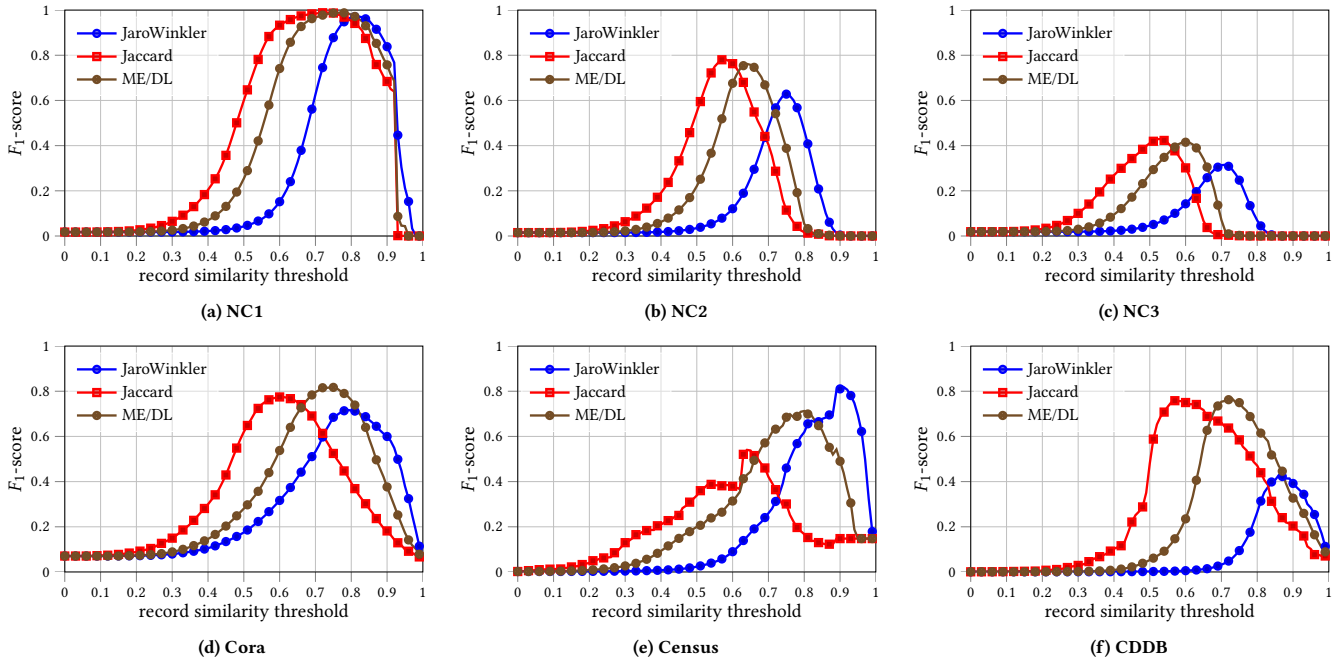


Figure 5: F_1-scores in relation to similarity thresholds for several similarity measures and test datasets

7 RELATED WORK

Almost all existing test datasets have been either artificially generated (e.g., by using an automatic generation tool) or been manually labeled. While artificial datasets have the obvious disadvantage of not containing actual real-life duplicates and errors (including outdated values), labeling duplicates manually is extremely expensive so that this approach is only an option for very small datasets. For example, the commonly used Cora, Census and CDDB datasets contain less than 10,000 records each (see Section 6.1). The essential shortcoming of an automatic labeling approach [32] is that the resulting gold standard is often not sound and biased towards the used algorithms, which can significantly distort evaluation results. One example of artificially created test data is the ERIQ dataset [31] which contains 100,000 records of customer data. Although this number is much larger than those of the manually labeled datasets, it is still small compared to typical big data applications. Further test datasets (manually labeled or artificially generated) can be found on the websites of the Hasso Plattner Institute[11], the University of Leipzig[19], and the Magellan project[20]. To the best of our knowledge, none of the existing test datasets provides precalculated similarity scores that help users to customize datasets on their basis.

Automatic test data generation tools can be divided into two classes: *Data synthetization* tools that generate all data values – including duplicates and errors – from scratch and *data pollution* tools that get a clean dataset as input and pollute it with duplicates, errors and inhomogeneities. Data synthetization tools, such as DBGen [14] or the Febrl Data Set Generator [3], are very efficient so that large datasets can be generated in short time [15]. However, since every data value is fictional, it is almost impossible to guarantee that the resulting values patterns are realistic. Data pollution tools, such as GeCo [6], TDGen [2], or DaPo [15], are the best option to generate test data with realistic value patterns because real-life data can be used as input. Moreover, if a broad spectrum of error types is supported they are nearly domain-independent. Except DaPo, however, existing pollution tools are strongly limited with respect to their scalability so that generating large datasets is either impossible or extremely time consuming [15]. A major problem that all these tools have in common is an appropriate simulation of outdated values and the complex error patterns that result from them.

Table 5: Previous usage of North Carolina voter data

	number of				cluster sizes†	
paper	records	attr.	clusters	dupl. pairs	avg	max
[18]	14,183	25	?	?	?	?
[5, 10]	8,261,838	19	8,110,137	155,469	2.02	6
[11]	200,000	6	100,000	200,000	2	2
[30]	5,000,000	4	3,500,840	3,331,384	4	5
[30]	10,000,000	4	6,625,848	14,995,973	7.7	10

†of non-singletons

Data of the NC voter register have been already used as test data in several works (see Table 5). Alas most of these uses are not fully documented and/or the provided links are outdated[21]. Thus, we cannot say exactly which data were used as input. The small size of the first dataset indicates that only a small portion of the voter register was used. The second dataset was created by Christen [5] in an earlier attempt to use the temporal changes of the voter data for generating test data with realistic outdated values. He regularly downloaded the current voter registration file on a bi-monthly basis over a time period of three years and combined these self-made snapshots after removing exact duplicates. However, instead of using the inherent gold standard

[19]https://dbs.uni-leipzig.de/en/research/projects/object_matching/benchmark_datasets_for_entity_resolution
[20]https://sites.google.com/site/anhaidgroup/useful-stuff/data
[21]ftp://www.app.sboe.state.nc.us [10, 11] and ftp://alt.ncsbe.gov/data/ [5]

provided by the NCID, he created it artificially by applying a rule-based duplicate detection approach. Thus, this gold standard is not guaranteed to be sound and biased towards algorithms using a similar detection approach.

Durham et al. [11] randomly selected 100,000 records from the voter register and generated polluted versions of these records by artificially introducing typos, semantic and phonetic errors. The last two datasets were created similarly by artificially polluting a randomly selected set of voter records with duplicates and errors using the GeCo tool [6]. Thus, these three datasets do not contain real-life errors and especially lack in realistic outdated values.

Except the last one, the sizes and duplicate distributions of these five datasets are nowhere near the numbers of the test dataset we generated in our study. Moreover, none of these authors enhanced their data with useful statistics, such as plausibility and heterogeneity scores, as we did in this study. Finally, to the best of our knowledge, we are the first who discuss the aspects of quality, usability and reproducibility in the context of test data for duplicate detection in such a depth.

8 CONCLUSION & FUTURE WORK

In this paper, we presented a large-scale dataset for evaluating duplicate detection algorithms and the approach behind its generation. Extracted from an historical voter register from North Carolina, our test dataset contains more than 120 million records and 640 million duplicate pairs making it uniquely suitable for evaluating duplicate detection at scale. Besides the dataset's size, our study focused on its quality, usability and reproducibility. The records' historical nature means that they contain many outdated values and – since data was often entered manually – also a large variety of other error types, such as typos, phonetic errors or confusions between attributes. While the data values themselves contain many errors, the underlying gold standard is largely error-free which is a mandatory requirement to ensure meaningful evaluation results. In general, its large size as well as its large number of different data errors makes the dataset perfectly suitable for users who want to customize their own datasets based on the requirements of their respective evaluation goals. To support such customizations, we equipped the individual records with similarity scores modeling their plausibility and heterogeneity. Finally, we integrated several mechanisms to ensure reproducibility when the dataset is growing with future updates. In summary, our approach enables generating large-scale test datasets with realistic errors including outdated values (which are hard to synthesize) and without the need for labeling duplicates manually (which is extremely labor-intensive).

Our plans for future work targets two different ways to extend our approach. First, we intend to generalize the procedure described here and apply it to historical corpora from other domains. This will provide the research community with large-scale test datasets beyond use cases that revolve around personal data. Second, we plan to combine our approach with a scalable data pollution tool, such as DaPo, to unite the strengths of having real outdated values and being able to inject additional errors at will. Our goal here is to increase the flexibility for customization and thereby facilitate generating test datasets geared for specific user demands. We think our presented work is useful to other researchers and we hope that our current line of research will pave the way for novel solutions that combine approaches using historical data with methods of data pollution in creative ways.

ACKNOWLEDGMENTS

We thank the staff of the North Carolina State Board of Elections for their extensive and valuable discussions on the structure and origins of the data.

REFERENCES

[1] North Carolina General Assembly. 2020. NC General Statutes. https://www.ncleg.gov/Laws/GeneralStatutesTOC. [Online; accessed 14-02-2021].
[2] Tobias Bachteler and Jörg Reiher. 2012. *Tdgen: A Test Data Generator for Evaluating Record Linkage Methods.* Technical Report wp-grlc-2012-01. German Record Linkage Center.
[3] Peter Christen. 2009. Development and User Experiences of an Open Source Data Cleaning, Deduplication and Record Linkage System. *SIGKDD Explorations* 11, 1 (2009), 39–48.
[4] Peter Christen. 2012. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer.
[5] Peter Christen. 2014. *Preparation of a Real Temporal Voter Data Set for Record Linkage and Duplicate Detection Research.* Technical Report. The Australian National University.
[6] Peter Christen and Dinusha Vatsalan. 2013. Flexible and Extensible Generation and Corruption of Personal Data. In *CIKM, USA*. 1165–1168.
[7] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. 2015. *Entity Resolution in the Web of Data.* Morgan & Claypool Publishers.
[8] AnHai Doan, Alon Halevy, and Zachary G. Ives. 2012. *Principles of Data Integration.* Morgan Kaufmann.
[9] Xin Luna Dong and Divesh Srivastava. 2011. *Big Data Integration* . Morgan & Claypool Publishers.
[10] Uwe Draisbach, Peter Christen, and Felix Naumann. 2020. Transforming Pairwise Duplicates to Entity Clusters for High-quality Duplicate Detection. *ACM J. Data Inf. Qual.* 12, 1 (2020), 3:1–3:30.
[11] Elizabeth Ashley Durham, Murat Kantarcioglu, Yuan Xue, Csaba Tóth, Mehmet Kuzu, and Bradley A. Malin. 2014. Composite Bloom Filters for Secure Record Linkage. *IEEE Trans. Knowl. Data Eng.* 26, 12 (2014), 2956–2968.
[12] Venkatesh Ganti and Anish Das Sarma. 2013. *Data Cleaning: A Practical Perspective.* Morgan & Claypool Publishers.
[13] Lise Getoor and Ashwin Machanavajjhala. 2013. Entity Resolution for Big Data. In *KDD*. 1527.
[14] Mauricio Hernández and Salvatore Stolfo. 1998. Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem. *Data Min. Knowl. Discov.* 2, 1 (1998), 9–37.
[15] Kai Hildebrandt, Fabian Panse, Niklas Wilcke, and Norbert Ritter. 2020. Large-Scale Data Pollution with Apache Spark. *IEEE Trans. Big Data* 6, 2 (2020), 396–411.
[16] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning.* ACM.
[17] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Dedoop: Efficient Deduplication with Hadoop. *Proc. VLDB Endow.* 5, 12 (2012), 1878–1881.
[18] Ioannis K. Koumarelas, Thorsten Papenbrock, and Felix Naumann. 2020. MDedup: Duplicate Detection with Matching Dependencies. *Proc. VLDB Endow.* 13, 5 (2020), 712–725.
[19] Luís Leitão and Pável Calado. 2011. Duplicate Detection through Structure Optimization. In *CIKM*. 443–452.
[20] Michael McDonald. 2019. United States Elections Project. http://voterlist.electproject.org/states/north-carolina. [Online; accessed 14-02-2021].
[21] David Menestrina, Steven Whang, and Hector Garcia-Molina. 2010. Evaluating Entity Resolution Results. *PVLDB* 3, 1 (2010), 208–219.
[22] MongoDB. 2019. MongoDB Documentation. https://docs.mongodb.com/.
[23] Felix Naumann and Melanie Herschel. 2010. *An Introduction to Duplicate Detection.* Morgan & Claypool Publishers.
[24] North Carolina State Board of Elections. 2021. Historical Voter Registration Snapshots. https://dl.ncsbe.gov/?prefix=data/Snapshots/. [Online; accessed 14-02-2021].
[25] North Carolina State Board of Elections. 2021. Public Data. https://s3.amazonaws.com/dl.ncsbe.gov/data/ReadMe_PUBLIC_DATA.txt. [Online; accessed 14-02-2021].
[26] North Carolina State Board of Elections. 2021. Voter Registration Data. https://www.ncsbe.gov/results-data/voter-registration-data. [Online; accessed 14-02-2021].
[27] Mateusz Pawlik, Thomas Hütter, Daniel Kocher, Willi Mann, and Nikolaus Augsten. 2019. A Link is not Enough - Reproducibility of Data. *Datenbank-Spektrum* 19, 2 (2019), 107–115.
[28] Julian Risch and Ralf Krestel. 2019. Measuring and Facilitating Data Repeatability in Web Science. *Datenbank-Spektrum* 19, 2 (2019), 117–126.
[29] Pramod J. Sadalage and Martin Fowler. 2012. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley Professional.
[30] Alieh Saeedi, Markus Nentwig, Eric Peukert, and Erhard Rahm. 2018. Scalable Matching and Clustering of Entities with FAMER. *CSIMQ* 16 (2018), 61–83.
[31] John R. Talburt. 2011. *Entity Resolution and Information Quality.* Morgan Kaufman Publ. Inc.
[32] Tobias Vogel, Arvid Heise, Uwe Draisbach, Dustin Lange, and Felix Naumann. 2014. Reach for Gold: An Annealing Standard to Evaluate Duplicate Detection Results. *J. Data and Information Quality* 5, 1-2 (2014), 5:1–5:25.

Industrial Paper

Path Indexing in the Cypher Query Pipeline

Jochem Kuijpers
contact@jochemkuijpers.nl
TU Eindhoven, Netherlands

George Fletcher
g.h.l.fletcher@tue.nl
TU Eindhoven, Netherlands

Tobias Lindaaker
tobias.lindaaker@neo4j.com
Neo4j, Sweden

Nikolay Yakovets
n.yakovets@tue.nl
TU Eindhoven, Netherlands

ABSTRACT

We investigate how a state of the art *path index* can be integrated into the Cypher query pipeline of the industrial Neo4j graph database. We identify the characteristics of practical use-cases where application of path indexes is beneficial to query evaluation and performance of index maintenance. Through in-depth empirical evaluation, we conclude that path indexes are most effective when used on *selective* patterns that allows the query planner to avoid high intermediate state cardinality and thus significantly accelerate query performance. As such patterns arise naturally in querying on real graphs, we can conclude that path indexes are a valuable method for improving the performance of graph database systems in practice.

1 INTRODUCTION

A common operation in graph databases is pattern query evaluation, i.e., looking for all matches of a query graph in a data graph [1]. This operation searches for sub-graphs in the data with a structure that is constrained by the query. A state of the art index on paths has been introduced in prior work [7, 17]. It has been shown that this index can be effective in accelerating query evaluation by multiple orders of magnitude and can be effectively maintained as the underlying data graph is updated [4, 12]. Current graph databases struggle with scalability, as graphs continue to grow in size and complexity [14]. Path indexes are a promising technique to help address query performance in practice.

In this short paper, we present experiences gained from the practical integration of a path index into the Cypher query pipeline of the Neo4j graph database management system. Cypher is a de facto industry standard query language for graph databases; Neo4j is one of the most popular and widely-deployed graph databases in industry [8]. We explore practical use-cases where path indexes can significantly improve query processing performance, and analyse scenarios when this query acceleration is achieved through an in-depth empirical evaluation. We conclude that path indexes are most effective when used on *selective* patterns that allow the query planner to avoid high intermediate state cardinality and thus significantly accelerate query performance. This result is not immediately obvious for contemporary graph database systems, and to our knowledge has not been observed before. As such patterns arise frequently in applications due to correlations in the structure of real world graphs, we can conclude that path indexes are indeed a valuable and practical method for scaling graph data management in industrial systems.

While our experiments were made using Neo4j, the results are immediately applicable to any graph database management system that implements Cypher or any other graph query language such as SPARQL or G-CORE that supports matching path patterns comprising a sequence of node and edge labels [1].

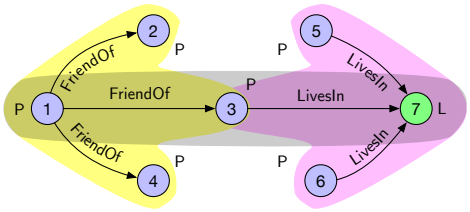


Figure 1: Example of a simple property graph. Nodes labeled *P* represent *Person* nodes. The node labeled *L* represents a *Location*. Shaded paths represent path indexes.

2 BACKGROUND

Property Graphs. Neo4j uses the *property graph* data model [1, 8]. A property graph is a graph where: every node can have an arbitrary number of labels; every relationship is directed between two nodes and has exactly one type; there may be multiple relationships of the same type between the same nodes, i.e. it is a multi-graph; and, every node and relationship can have an arbitrary number of associated attribute-value pairs. As an example, Figure 1 shows a property graph (suppressing attribute-value pairs) representing a small social network.

Cypher Query Language. Cypher is a declarative graph query language that is loosely based on SQL [8]. It contains familiar SQL keywords such as `WHERE` that function essentially the same by allowing users to apply predicates to filter the results of the query. However, in Cypher, the primary way to retrieve data is using the `MATCH`-clause. Such a clause contains one or more pattern expressions. A pattern expression is an alternating sequence of nodes and relationships, starting and ending with a node. Nodes are expressed using parentheses while relationships are expressed as arrows. Query variables are declared by their inclusion in one or more pattern expressions and can be used in other clauses. For example, `(x:Person)` matches all nodes `x` with label `Person` and `(p:Person)-[r:Lives_In]->(c:City)` matches all directed relationships `r` from nodes `p` to nodes `c`, with labels `Lives_In`, `Person`, and `City`, resp.

Path Patterns. A *path pattern* is a sequence of alternating node labels and relationship patterns starting and ending with a node label. A *relationship pattern* contains both a relationship type and its direction (either forward, $\rightarrow$, or reversed, $\leftarrow$). E.g., given the node labels $\{A, B\}$ and the relationship type R, the following notation describes a path pattern of length 2: $\langle A, (R, \rightarrow), B, (R, \leftarrow), B\rangle$, counting relationships to determine the length.

A path pattern describes a set of constraints that can be applied to paths of the same length. Given a k-length path pattern

$\langle N_0, (R_1, D_1), \ldots, (R_k, D_k), N_k \rangle$, then a path $\langle n_0, r_1, \ldots, r_k, n_k \rangle$ in the graph satisfies the pattern if and only if all nodes n_i have a N_i label for $0 \leq i \leq k$ and all relationships r_i are of type R_i and direction D_i for $1 \leq i \leq k$. As an example, the pattern $\langle P, (FriendOf, \rightarrow), P, (LivesIn, \rightarrow), L \rangle$ would yield one query result on the graph in Figure 1.

Prior Work on Path Indexing. Querying graph databases using indexes is a complex field of study. A detailed contemporary survey of existing techniques can be found in [1]. The *k-path index*, introduced by Fletcher et al. [7, 13, 16, 17], demonstrated that significant orders-of-magnitude query performance improvements are possible with path indexing. Here, the intuition is to index all, or a selected subset of all, paths of length up to k, where k is the count of relationships in the indexed paths. Fletcher et al. built the k-path index by concatenating edges in the graph and storing the resulting paths in a relational database [7]. The resulting table was indexed and used to speed up path queries.

Sumrall et al. [17] engineered a k-path index directly implemented using a B$^+$-tree and demonstrate the potential for accelerated query processing. Sumrall also studied how one could index paths with specific patterns (e.g., as given in a workload) rather than all paths of length k.

Persson [12] utilized the path index concept to speed up dense network data retrieval by indexing all paths of length 1. This index was called a *Shortcut Index*. Persson intentionally limited his work to indexes with a single relationship to avoid expensive maintenance computations on graph updates, which could not be afforded in the described use-case.

De Jong [4] demonstrated how indexes can be more efficiently maintained by maintaining sub-patterns as separate indexes. This allows for a significant speed-up of index maintenance, at the expense of increased storage overhead.

Boncz et al. [3] and Luo et al. [10] give a broader overview of graph query processing with indexes, both in contemporary systems and in research, as well as the role of query selectivity in effective graph query processing. To our knowledge, ours is the first study of path indexing in industrial systems.

Path Indexes. A *path index* is a data structure that indexes all paths in the data graph which satisfy a chosen path pattern. This path pattern is called the *indexed pattern*. The paths are not stored directly in the path index, instead a sequence of references is stored. For every indexed k-length path $\langle n_0, r_1, \ldots r_k, n_k \rangle$, the index stores references to the nodes and relationships of the matching paths as a sequence of identifiers: $\langle n_0^{id}, r_1^{id}, \ldots, r_k^{id}, n_k^{id} \rangle$, where n_i^{id} and r_i^{id} are the i-th identifiers of nodes and relationships in the path respectively.

These sequences of references are converted into a single key by concatenating the fixed-width identifiers (8 bytes each), which are stored in a B$^+$-tree. This allows logarithmic-time location, insertion and deletion of entries in the index. Since the identifiers are concatenated, the B$^+$-tree also supports prefix searches. Given the first m elements of a path, we can locate the first entry that starts with this prefix, and scan all paths that match the prefix in linear-time with respect to the number of results returned. The worst-case space complexity for the index is $O(E^k)$ where E is the number of edges in our graph and k the length of the path pattern that is indexed. While the size of the index is linear to the number of identifiers in the key, each index is defined for a fixed size key which keeps the size bounded. It is also worth noting that when indexes are chosen to represent selective patterns, the size of the index will naturally remain small.

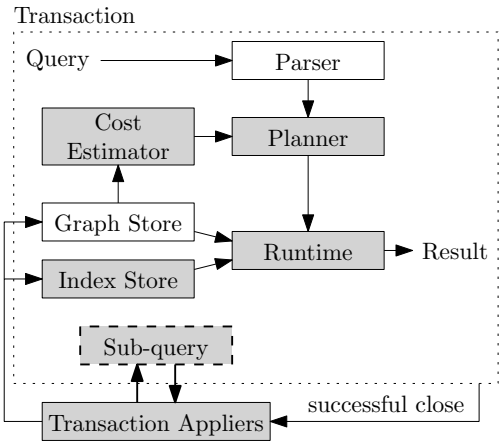


Figure 2: An overview of the query pipeline architecture. Shaded boxes represent our changes, while dashed outlines represent a new component.

The shaded parts on Figure 1 represent path indexes. An index on $\langle P, (FriendOf, \rightarrow), P, (LivesIn, \rightarrow), L \rangle$ (shaded grey) contains just the single path through nodes $\langle 1, 3, 7 \rangle$, an index on $\langle P, (FriendOf, \rightarrow), P \rangle$ (yellow) will contain the paths through nodes $\{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle\}$ and an index on $\langle P, (LivesIn, \rightarrow), L \rangle$ (purple) will contain the paths through nodes $\{\langle 3, 7 \rangle, \langle 5, 7 \rangle, \langle 6, 7 \rangle\}$.

3 USING PATH INDEXES

Our Extensions. We next discuss the modifications and extensions to the query pipeline architecture necessary to support path indexes (Figure 2). Firstly, path index operators have been added to the *Planner* component which can scan or selectively read paths from path indexes. This also requires new costing heuristics in the *Cost Estimator* for these operators. The new operators are also implemented in the *Runtime* component as runtime-specific operators that perform the read operations on the path index store. Further the *Index Store* was modified to hold the new type of path index.

When a transaction is committed, the update commands are translated into paths that have to be added to or removed from the path indexes through sub-queries on the path patterns of those indexes. These sub-queries are inserted into the query pipeline as normal and might use other path indexes to resolve the query, depending on the context. Full details can be found in the extended report [9].

4 IMPLEMENTATION

We next discuss integrating the path index into the database code-base. We will start with an overview of all components that required modifications in Section 4.1. Section 4.1.1 describes query-based path index maintenance. Then Section 4.1.2 provides information on path index initialization.

4.1 Modified Pipeline Components

The implementation of our path index into the query pipeline required modifications in several components. Figure 2 shows an overview of the components, where a shaded background means the component required some modification. The dashed block "Sub-query" represents a new component.

The cost estimator is extended to estimate a cost for path index operators. We re-used the existing cardinality estimator

due to scoping constraints which assumes that all filtering and combining operations behave according to global data statistics.

4.1.1 Query-based path index Maintenance. De Jong [4] observed that, as a consequence of the policy in the Neo4j database to never allow the deletion of a connected node, we only ever need to look for relationship updates in the graph in order to update the path index. De Jong describes two methods for translating graph updates into path index updates.

(1) *Traversal-based translation:* starting from the updated relationship, traverse the indexed pattern on the data graph and make note of all the paths encountered. If the relationship was added, then add all these paths to the index. Otherwise, remove all these paths from the index.
(2) *Self-maintaining translation:* maintain a path index for all sub-patterns of the index pattern. Some of the sub-patterns need to be reversed in order to do prefix-scans on these indexes. Then, when handling an updated relationship, simply do a prefix search on the largest index that contains the changed relationship in both directions, and combine the two resulting sets created by these actions. Then add or remove these paths from the index.

The first method is a naive graph traversal. The second method requires all sub-patterns to be indexed. We introduce the following maintenance method, which uses the query pipeline itself to find the most effective way to search for updated paths:

(3) *Query-based translation:* query the index pattern with an additional predicate that the modified relationship must be part of the resulting paths. This query then returns all paths through the updated relationship. We can then add or remove these paths from the index.

The last approach is far more flexible in terms of which pattern indexes are allowed to exist, compared to self-maintaining translation. However, it requires executing new queries while processing transactions. This broke some assumptions about the way transactions are handled in Neo4j, namely one transaction per execution thread. As a result, our prototype does not support concurrent updates. Another issue was that we needed to by-pass the query cache, otherwise we had no control over which indexes would be used in the maintenance queries.

When a relationship is modified, added or removed in the graph, this could mean entries need to be added or removed from the path indexes. To find out which paths have been changed, a query is executed that contains the pattern of the index and an additional constraint that the relationship in the pattern must match the updated relationship from the transaction. This is described in Algorithm 1.

There are some important things to consider. Firstly, Neo4j 3.5[1] binds a transaction along with its transaction state to the thread that opened the transaction. That means that, while we are applying the outer transaction, the inner query we want to execute is filtered through the transaction state during maintenance. As a work-around to this behavior, we store the transaction state of the outer transaction and reset it for the maintenance queries. After processing the updates, we restore the old transaction state such that other Neo4j operations are not affected.

Further, since we are in the middle of applying a transaction, some of the path indexes might not be up-to-date while other path indexes may already have been updated. We introduce a sort order of path indexes by length, small-to-large, to ensure

[1]Transactions no longer bind to threads in Neo4j 4.0

Algorithm 1 Maintenance

Input Modified relationship r in graph G.
1: b := the label of the start node of r
2: e := the label of the end node of r
3: t := the type of r
4: I := a list of path indexes with patterns that contain `...(:b)-[:t]->(:e)...`
5: Sort I by pattern length, ascending.
6: T_{old} := the committed transaction state.
7: Reset the transaction state.
8: **if** r is a removed relationship **then**
9: **for** all *index* in I **do**
10: P := the pattern of *index* containing relationship r
11: R := Query(P, G)
12: Remove all entries R from *index*.
13: Process all other transaction appliers for r.
14: **if** r is an added relationship **then**
15: **for** all *index* in I **do**
16: P := the pattern of *index* containing relationship r
17: R := Query(P but avoid using *index*, G)
18: Add all entries R to *index*.
19: Set the transaction state to T_{old}.

Algorithm 2 Index initialization

Input index pattern P, data graph G
Output initialized index I
1: I := a new path index
2: *Result_Iterator* := Query(P, G)
3: **while** *Result* := *Result_Iterator.next* **do**
4: Add *Result* to I

that any maintenance query plan for a path of length k will itself only include path indexes of lengths smaller than k, which by then have already been updated. When we remove a relationship from the graph, we want the maintenance query results to still include it so we know which paths to remove from the index. That is why the query for removals is done before modifying the underlying data. For relationship additions, the reverse holds. We want to include it in the maintenance query results in order to add these paths to the index, therefore we must query these after the underlying data has been updated.

Similar maintenance steps can be applied for node label updates. Node additions and removals can be ignored, as those are only allowed for disconnected nodes, making it impossible to affect path index maintenance.

4.1.2 Initialisation. A small but important aspect is being able to create indexes on existing data: index initialization. This is done by querying the pattern on the existing data graph and adding the result set to the new index in a single transaction. Other indexes that have already been initialized may be used at this point. Index initialization thus follows the simple procedure described in Algorithm 2.

Sumrall [16] proposed constructing a B^+-tree directly from a sorted list of query results in order to speed up the B^+-tree construction, though this was not practical to achieve in our implementation since the B^+-tree memory layout is abstracted in the code base. As index initialization was not the primary focus of this study, we used our more naive approach, which increases the one-time construction cost of any path index.

5 EXPERIMENTAL SETUP

Baseline Planner Extension. The planning model used by Neo4j is node-centric. There exist a number of ways to selectively scan or seek nodes by their node labels and indexes exist on node properties, but there are few ways to selectively scan or seek based on relationship types. Our path index implementation will have these abilities. Therefore we apply an extension to the baseline planner which includes an operator that scans relationships by type. It is introduced with the same cost heuristics as the most similar node-based operator. Our path index query plans are compared with this extended baseline.

Hardware and Software. Our experiments were performed on a server with four Intel Xeon E5-4610 v2 CPUs running at 2.30GHz, 500GB of DDR3 RAM at 1600MHz. We used its 260GB NVMe SSD for data storage. The server ran Ubuntu 16.04.3 LTS and the Oracle Java (TM) SE Runtime Environment (version 1.8.0-151). We prototype on the Neo4j 3.5 community code base [11].

Methodology. Our experiments ran with a pre-allocated heap of 100GB. Each experiment ran until running time converges, which indicates that hot code paths were optimized by the JVM. Then we ran the experiment five times, triggering a garbage collection cycle between each run and flushing the data from memory without restarting the JVM as this would lose hot code path optimizations. We then discarded the highest and lowest running time and averaged the remaining three results. For data set sizes, we summed the total data file sizes on disk. Path index stores were measured separately and transaction logs were excluded.

Datasets. We use four data sets in our experiments: two synthetically generated and two real-world data sets. The first synthetic data set is referred to as the *correlated* data set, as it has high structural correlation. It has 125K nodes and 12.6M relationships and was created by interconnecting 25 000 copies of the same path, making it a highly selective pattern. The second synthetic data set is referred to as the *independent* data set as there are no structural correlations in the connections between nodes. It has 250K nodes and 5M relationships. The first real-world data set is the YAGO data set [15], containing 77M nodes and 100M relationships. The second real-world dataset is the GeoSpecies data set [5], containing 225K nodes and 1.5M relationships. Coming from distinct application domains, these graphs allow us to gain practical insights into the robustness of our methods.

6 RESULTS

Our hypothesis is that our path index is not suited for application on high-cardinality path patterns, since the worst-case space requirement is exponential in path length. Indeed, we observe that path indexes are especially useful for highly selective paths on correlated data, as the cost of the path index is very low compared to the computation of intermediate state that can be *skipped* by using the index.

To test this hypothesis, we first run two controlled scenarios of queries on highly correlated and uniformly distributed synthetic data sets. We generate our own data sets, rather than using a benchmark such as LDBC SNB [6], so as to finely control the structure of the data. This was sufficient for our goals here, but we note that off-the-shelf generators such as gMark could also have been used [2]. Then, we verify our findings by applying our path index to two real-world scenarios: a selective, correlated path query and a high-cardinality path query.

Finally, we show that path indexes can be applied to index maintenance in some cases, and the effect of selective, correlated index path patterns.

6.1 Synthetic Query Benchmarks

Our first experiments show that choosing the right path index can significantly improve query performance. The available indexes on this data set are described in Figure 3 (Correlated synthetic). The query pattern matches that of the full index. The result of this query when planned with different indexes can be seen in Figure 4 (Correlated synthetic). The full index is clearly the best as it essentially pre-computes the answer. However, sub-index S_1 has similar performance for a smaller index, which may offer more re-use capabilities.

The second experiment, illustrated in Figure 4 (Independent synthetic), shows the same technique applied to a uniformly distributed data set. The indexes available here are described in Figure 3 (Independent synthetic). Because there is no selective, structural correlation in this data set, the query produces many more results. Indexing these results, even in sub-indexes, provides no significant speed improvement.

Both of these experiments show dependency between the running time and the maximum intermediate cardinality. This indicates that indexing selective patterns can significantly reduce the maximum intermediate cardinality during query evaluation, and thus the running time of the query.

6.2 Real-world Query Benchmarks

After findings on synthetic datasets, we applied our technique to real-world data sets. Our first dataset, YAGO [15], has a file containing query workloads. We used the cardinality estimation model of the Neo4j planner that assumes independence between elements to find the query that was most mis-predicted, as our assumption was that this query would be highly correlated, since this is exactly the type of query that will yield mis-predictions by this cardinality estimation model.

We then applied path indexes that matched parts of the query to speed up query evaluation [9]. The heuristic cost estimator we used was built on the assumption of independence. For the YAGO experiments, the resulting query plans were of insufficient quality. We have manually created better query plans to show what an improved cost estimator could achieve with our indexes. The indexes are described in Figure 3 (YAGO dataset) and the benchmark results are shown in Figure 4 (YAGO).

The full index on the query pattern significantly speeds up query evaluation time compared to our manually optimized baseline. The plans using smaller sub-indexes further improve performance, even though it requires more steps to fully answer the query in these plans. This can be explained by the reduction of intermediate cardinality in later stages of the operator tree. The high cardinality of the S_2 and S_3 plans is caused by the first node scan operator which is reduced early on in the execution of the query plans, this explains the faster execution time despite the initially higher cardinality compared to the F plan. The path index efficiently produces the full path on this reduced state, achieving fast total execution times.

We have also applied this to another data set with a less selective query, expecting our path index would not be able to speed up query evaluation performance as much. And indeed, our experiment results (shown in Figure 3 and Figure 4 under

Correlated synthetic

Name	Indexed pattern	Cardinality	Size (MB)
G	–	–	413.97
F	`(:A)-[:X]->(:A)-[:X]->(:A)-[:Y]->(:B)-[:X]->(:A)`	25 000	3.92
S_1	`(:A)-[:X]->(:A)-[:X]->(:A)-[:Y]->(:B)`	25 000	3.17
S_2	`(:A)-[:X]->(:A)-[:Y]->(:B)-[:X]->(:A)`	25 000	3.17
S_3	`(:A)-[:X]->(:A)-[:X]->(:A)`	12 524 000	970.56
S_4	`(:A)-[:X]->(:A)-[:Y]->(:B)`	25 000	2.39
S_5	`(:A)-[:Y]->(:B)-[:X]->(:A)`	6 274 500	471.59
S_6	`(:A)-[:X]->(:A)`	6 299 500	364.95
S_7	`(:A)-[:Y]->(:B)`	6 274 500	250.27
S_8	`(:B)-[:X]->(:A)`	25 000	1.55

Independent synthetic

Name	Indexed pattern	Cardinality	Size (MB)
G	–	–	171.24
F	`(:A)-[:V]->(:B)-[:W]->(:C)-[:X]->(:D)-[:Y]->(:E)`	862 345	97.92
S_1	`(:A)-[:V]->(:B)-[:W]->(:C)-[:X]->(:D)`	280 050	33.97
S_2	`(:B)-[:W]->(:C)-[:X]->(:D)-[:Y]->(:E)`	295 337	35.55
S_3	`(:A)-[:V]->(:B)-[:W]->(:C)`	111 532	10.42
S_4	`(:B)-[:W]->(:C)-[:X]->(:D)`	102 812	9.72
S_5	`(:C)-[:X]->(:D)-[:Y]->(:E)`	129 410	8.70
S_6	`(:A)-[:V]->(:B)`	40 039	2.45
S_7	`(:B)-[:W]->(:C)`	40 227	2.47
S_8	`(:C)-[:X]->(:D)`	40 613	1.97
S_9	`(:D)-[:Y]->(:E)`	40 220	1.84

YAGO dataset

Name	Indexed pattern	Cardinality	Size (MB)
G	–	–	20 947.05
F	`(a)-[w]->(b)-[v]->(c)-[x]->(d)-[y]->(e)-[z]->(f)`	2 320	0.45
S_1	`(a)-[w]->(b)-[v]->(c)-[x]->(d)`	7	< 0.01
S_2	`(b)-[v]->(c)-[x]->(d)-[y]->(e)`	12 323	1.58
S_3	`(c)-[x]->(d)-[y]->(e)-[z]->(f)`	366	0.01

GeoSpecies dataset

Name	Indexed pattern	Cardinality	Size (MB)
G	–	–	117.99
F	`(a)-[x]->(b)<-[y]-(a)-[x]->(b)`	334 126	32.13
S	`(a)-[x]->(b)`	24 814	1.54

Figure 3: The available indexes on the benchmarked datasets with their cardinality and storage size. Here, G denotes the size of the whole graph, F the index for the full path, and S indexes for sub-paths.

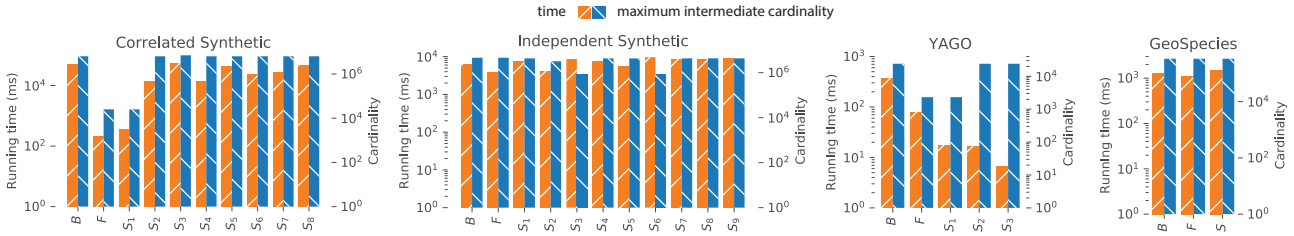


Figure 4: Benchmark results on the data sets. B denotes the baseline, F indexing the full path, and S indexing sub-paths.

Name	Relationship addition		Relationship deletion		Average speed-up
	Full index	Sub index	Full index	Sub index	
None	0.436 ms	–	0.836 ms	–	–
Sub_1	0.301 ms	0.266 ms	0.824 ms	0.570 ms	≈ 0.65×
Sub_2	0.368 ms	0.333 ms	0.855 ms	0.586 ms	≈ 0.59×
Sub_3	0.288 ms	–	0.675 ms	–	≈ 1.32×
Sub_4	0.277 ms	0.242 ms	0.648 ms	0.714 ms	≈ 0.68×
Sub_5	7 885.829 ms	0.536 ms	7 919.113 ms	1.025 ms	≈ 0.00×
Sub_6	0.184 ms	–	0.489 ms	–	≈ 1.89×
Sub_7	7 110.481 ms	0.561 ms	7 142.197 ms	0.734 ms	≈ 0.00×
Sub_8	0.219 ms	–	0.443 ms	–	≈ 1.92×

(a) Correlated data

Name	Relationship addition		Relationship deletion		Average speed-up
	Full index	Sub index	Full index	Sub index	
None	0.362 ms	–	0.824 ms	–	–
Sub_1	412.406 ms	0.269 ms	419.108 ms	0.742 ms	≈ 0.00×
Sub_2	94.959 ms	0.297 ms	94.883 ms	0.953 ms	≈ 0.01×
Sub_3	0.303 ms	–	0.808 ms	–	≈ 1.07×
Sub_4	152.848 ms	0.323 ms	152.430 ms	1.006 ms	≈ 0.00×
Sub_5	36.943 ms	0.280 ms	42.162 ms	0.570 ms	≈ 0.01×
Sub_6	0.245 ms	–	0.596 ms	–	≈ 1.41×
Sub_7	0.334 ms	–	0.930 ms	–	≈ 0.94×
Sub_8	48.862 ms	0.204 ms	34.820 ms	17.250 ms	≈ 0.01×
Sub_9	0.526 ms	–	1.340 ms	–	≈ 0.64×

(b) Independent data

Figure 5: Results of the maintenance experiment on correlated (left) and independent (right) data. The rows show the amount of time required to update the index, given the presence of a sub-pattern index named in the left-most column.

GeoSpecies) show that, because the result cardinality is the highest cardinality in the query evaluation, our path index was not able to *skip over* large intermediate cardinalities, and thus no real performance gain was achieved.

6.3 Maintenance Using Sub-Indexes

Not only can sub-pattern indexes provide performance benefits during query execution. As De Jong [4] showed, sub-patterns can also be used to speed up the maintenance of the full index. Where the *self-maintaining translation* introduced by De Jong exhaustively provides all sub-patterns such that no data has to be read from the graph, our approach simply defers this decision to the query planner, as maintenance is performed using a specific query on the indexed pattern. This allows us to pick an arbitrary set of path indexes, which may then also be used to speed up maintenance when applicable.

In this experiment, we first look at the performance benefits on our synthetic correlated data set for index maintenance, as we provide one of the sub-pattern indexes from Table 3 alongside the *Full* index. The graph is updated to remove one of the Y-labeled relationships in a transaction, after which this same relationship is added again in a new transaction. Fig. 5 (a) shows the results of this experiment. The first row contains the maintenance performance of just the *Full* index and the subsequent rows contain the

performance of using a maintenance plan that includes the sub-pattern index. Further, the sub-pattern index itself may also need to be maintained, thus these measurements are also included. For this experiment, the query planner is forced to use a plan that uses the sub-pattern index for the *Full* index maintenance. This sometimes results in a slower maintenance plan as only some query plans are considered. We may assume that the planner would not use the sub-index for maintenance in that case, if given the choice, though the figures give an indication of the effect on maintenance of the sub-index. The average speed-up reported is the factor of performance increase of both maintenance operations for the removal and addition of a relationship.

Interestingly, both Sub_1 and Sub_4, the indexes that provided the most performance increase during query execution, do not speed up the maintenance operations of changes to this specific relationship. Sub_3 provides a moderate performance increase for maintenance computations, while it was the worst performing index in the previous query execution experiment.

We then perform the same set of transactions, by removing an Y-labeled relationship in a transaction and adding it in another transaction, leading to index maintenance on the synthetic independent data set. We observe that similar modest speed improvements can be achieved the sub-pattern indexes on the full index maintenance in Fig. 5 (b), while the forced plans for some sub-indexes perform considerably worse as well.

7 LESSONS LEARNED

Indexing paths in graphs often makes sense in practice. Since the number of (potential) paths in a graph grows exponentially with path length, it might seem too prohibitive (wrt. worst-case space complexity) to index path patterns in large graphs. In this work, on the contrary, we found out that, in practice, indexing *strategically-chosen* path patterns can, in fact, greatly improve query performance in both synthetic and real datasets at a *small* storage overhead. The practicality of this approach is underscored by our integration of our work in the Neo4j system.

Patterns with high structural correlation are most beneficial to index. Patterns where there is high correlation in the connections between the nodes in the data will result in a relatively low number of paths matching the pattern. The number of edges that have to be explored to match the same pattern through direct traversal of the graph would be substantially larger. In such situations the cardinality of the intermediate result is substantially larger than cardinality of matches to the whole pattern. Our experiments show that when correlated patterns are indexed, this high cost of computing these intermediate results of high cardinality is avoided. Furthermore the size of the index for such a highly selective pattern over correlated data is small as well. This turns out to be a sweet spot for path indexes, where the benefit of the index is high and the overhead of the index is low.

In contrast, for patterns matching uncorrelated data, the size of the index is proportional to the cardinality of the intermediate result, which in many cases grow exponentially in the size of the underlying graph. In these cases we experience not only a prohibitively large storage overhead for the index, but also no tangible performance benefit, since enumerating the paths from the index is proportional to enumerating the paths by direct traversal of the underlying graph.

Patterns to be indexed should be chosen with care. Hence, one should take advantage of structural correlations which naturally occur in graphs in order to choose path patterns that have (1) low cardinality and (2) help to cut down on the cardinality of intermediate results during the evaluation of queries in the workload. Finding path patterns that satisfy both (1) and (2) is not trivial and is ultimately a constrained optimization on the given workload and usable storage.

8 CONCLUSIONS

We have reported on our practical experiences integrating a state of the art path index into the query processing pipeline of Neo4j, a popular industrial graph database. Through extensive empirical study, we found that selective path indexes can greatly accelerate query evaluation performance. This is especially true in those cases where the query engine would otherwise require computations on large intermediate state to arrive at a relatively small result set. In these scenarios, which arise commonly in practice due to correlations in the structure of real world graphs, we have shown that even though path indexes in the worst case require exponential storage, these selective path indexes can be very small relative to the total graph size. These are optimal scenarios for path indexes since the path index is able to provide a significant performance improvement with a low space overhead. Our extended report [9] contains further details and results, such as technical aspects of the integration into Neo4j, the technical challenges encountered, and the engineering lessons learned.

Looking ahead, there are several interesting directions for further research. We close by indicating two of these: (1) investigate more deeply query planning in the presence of path indexes, including cardinality estimation and costing techniques for path indexes; and, (2) study methods for selecting which patterns to index, balancing space costs and performance benefit, e.g., with respect to a given query workload.

REFERENCES

[1] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying graphs.* Morgan & Claypool Publishers.

[2] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869.

[3] Peter A. Boncz, Orri Erling, and Minh-Duc Pham. 2014. Advances in Large-Scale RDF Data Management. In *Linked Open Data - Creating Knowledge Out of Interlinked Data - Results of the LOD2 Project.* LNCS, Vol. 8661. 21–44.

[4] Niels de Jong. 2019. *MAGPIE (a Maintainable Graph Pattern Indexing Engine): Towards a versatile path index for the industrial graph database.* Master's thesis. Eindhoven University of Technology, Eindhoven, The Netherlands.

[5] Peter DeVries. 2009. The GeoSpecies Knowledge Base ontology. http://rdf.geospecies.org/geospecies.rdf.gz. Accessed in March 2019.

[6] Orri Erling et al. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD.* 619–630.

[7] George Fletcher, Jeroen Peters, and Alexandra Poulovassilis. 2016. Efficient regular path query evaluation using path indexes. In *EDBT.* 636–639.

[8] Nadime Francis et al. 2018. Cypher: An evolving query language for property graphs. In *SIGMOD.* 1433–1445.

[9] Jochem Kuijpers. 2020. *Path Indexing in the Cypher Query Pipeline.* Master's thesis. Eindhoven University of Technology, Eindhoven, The Netherlands.

[10] Yongming Luo et al. 2012. Storing and Indexing Massive RDF Datasets. In *Semantic Search over the Web.* Springer, 31–60.

[11] Neo4j Inc. 2019. Neo4j 3.5 source code. https://github.com/neo4j/neo4j/tree/3.5. Accessed in January 2020.

[12] Anton Persson. 2016. *The Shortcut Index.* Master's thesis. Lund University, Lund, Sweden.

[13] Jeroen Peters. 2015. *Regular path query evaluation using path indexes.* Master's thesis. Eindhoven University of Technology, Eindhoven, The Netherlands.

[14] Siddhartha Sahu et al. 2019. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB Journal* (2019).

[15] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: A Core of Semantic Knowledge. In *WWW.* 697–706.

[16] Jonathan M. Sumrall. 2015. *Path indexing for efficient path query processing in graph databases.* Master's thesis. Eindhoven University of Technology, Eindhoven, The Netherlands.

[17] Jonathan M. Sumrall et al. 2016. Investigations on path indexing for graph databases. In *PELGA @ Euro-Par.*

Industrial Paper

A Deep Learning Architecture for Audience Interest Prediction of News Topic on Social Media

Ciprian-Octavian Truică[*†]
Faculty of Automatic Control and Computers
University Politehnica of Bucharest
Bucharest, Romania
ciprian.truica@upb.ro

Elena-Simona Apostol[*†]
Faculty of Automatic Control and Computers
University Politehnica of Bucharest
Bucharest, Romania
elena.apostol@upb.ro

Teodor Ștefu[†]
Faculty of Automatic Control and Computers
University Politehnica of Bucharest
Bucharest, Romania
teodor.stefu@cti.pub.ro

Panagiotis Karras
Department Computer Science
Aarhus University
Aarhus, Denmark
panos@cs.au.dk

ABSTRACT

Personalized social media offer communication opportunities that mass media could not afford, yet also raise novel challenges. A prime challenge arising from this shift in digital communication is to detect topics and events of interest. In this paper, we propose and deploy a novel Deep Learning architecture that predicts if a *news topic* becomes viral by analyzing social media diffusion and audience interest in current *news events*. The proposed solution: (i) analyzes news articles, (ii) extracts associated topics and events, (iii) matches the topics and events to filter and extract developing topics, (iv) extracts current events from Twitter and matches them to the filtered news topics, and (v) predicts audience interest in news topics using Twitter likes and retweets. We employ several feature engineering techniques to improve prediction by integrating user metadata into the training set. In our experiments, we correlate two datasets collected over several months in the same time period. The first dataset contains news articles collected from different news venues, while the second one contains tweets regarding the news. The experimental results from our real-world deployment prove that the proposed system achieves high accuracy when integrating *influencers* metadata and the day of the week. Thus, proving that the *news topics* virality prediction is improved under the assumptions that spreaders and the day of the week play a huge role in information diffusion.

KEYWORDS

audience interest prediction, social media, news diffusion, topic modeling, event detection, neural networks

1 INTRODUCTION

The internet age has brought new ways of sharing information that changed the way the general public consumes media content. Physical newspapers moved to a digital form in a few years. With this migration to the virtual world, the number of news sources increased, and consumers gained a wide variety of options from where to choose their daily news using their preferred content.

Users have two methods at their disposal to stay up to date with current *news events*:

(1) buy a subscription to news publisher that provides in-app or email news content, or
(2) follow the content of news outlets or *influencers* on social media [30], where *influencers* are users that manage to sway a target audience.

An emerging and growing social media platform is Twitter. Twitter provides users a venue where they can share their thoughts, discuss, and forward various kinds of information [29]. The subjects are diverse, from daily local events to important global issues. The ever-growing number of users around the world tweeting makes Twitter a valuable source for real-time information. Thus, many consumers spend time on their news feeds to catch up with everything around them.

News outlets and publishers also embraced Twitter to update followers by regularly posting tweets that contain short news headlines and including direct links to full articles. Thus, news outlets reach large audiences and increase their reader base. Yet, to predict the audience's interest in a news topic, we should determine the underlying social structures and relationships between Twitter users [14], and utilize the network graph structure modelling the relationships between members of different social groups [18]. Nodes in a group's center are called *influencers* as they have a huge role in spreading the information. Nodes that like or retweet content are known as *spreaders*, as they propagate the information further in the network.

Our proposed solution is motivated by the recent negative impact viral Fake News has on society at large and the current need for systems that can help in stopping the viral spreading of such news. Knowing the veracity, determining the propagation patterns, and predicting the influence of news articles in social media, new strategies for mitigating harmful misinformation can be developed. Thus, the main objective of this paper is to determine if a news topic becomes viral on social media. We tackle this issue by predicting the audience in order to facilitate the development of new network immunization strategies.

The research questions we are trying to answer are:

(Q1) Do the current events presented in mass media also gain traction on social media?
(Q2) Does the diffusion of news articles on social media influence the trending topic of current events?

[*]Corresponding Author
[†]These authors contributed equally to the paper

(Q3) Can we predict the audience interest in a news topic by analysing the retweets and likes received by news articles belonging to the topic on social media?

(Q4) Does the network-related metadata improve the audience interest prediction?

To answer *(Q1)* and *(Q2)*, we first extract the *news topics* using Non-Negative Matrix Factorization (NMF) [2] and the *news events* using Mention-Anomaly-Based Event Detection (MABED) [15]. Second, we correlate the *news topics* and *news events* to extract *trending news topics* by employing the cosine similarity. Finally, we extract *Twitter events* also using MABED. For answering *(Q1)*, we analyze the correlation results of *trending news topics* → *Twitter events*. Whereas, for answering *(Q2)*, we analyse the reverse correlation results, i.e., *Twitter events* → *trending news topics*.

To answer *(Q3)*, we consider that the correlated < *trending news topics*, *Twitter events* > pair reveals insights in the audience interest through likes and retweets. Thus, we train two Deep Learning models on the *Twitter events* to predict their likes and retweets on Twitter. The high accuracy of the models proves that we can correctly predict the audience interest in *trending news topics* using social media information and determine if they become viral.

Finally, to answer *(Q4)*, we enhance the training dataset with metadata containing information regarding each tweet's author and its number of followers as well as the day of the week when the tweet was posted online. We base this approach on two assumptions. First, that *influencers* (users with a high number of followers) have a huge role in spreading the information and making news articles topics viral. Second, that user behaviour posting patterns, as well as media consumption, are influenced by the day of the week, as also empirically proven in [3]. We retrain our Deep Learning models and manage to obtain better accuracy results, thus showing that the assumptions stand for our use case.

We use Twitter's graph structure to extract events and to predict the audiences' interest by analyzing the textual and metadata content (e.g., retweets, likes, user followers, etc.) of each post. Thus, we propose and deploy a novel architecture that incorporates topic modeling, event detection, and Deep Learning classification. Our system:

(1) analyzes news articles,
(2) extracts their topics and associated events,
(3) matches these topics and events to filter and extract developing topics,
(4) extracts current events from Twitter and matches them with the filtered news topics, and
(5) predicts audience interest in the news topics using Twitter likes (formally known as favorites) and retweets by enhancing the dataset with metadata, i.e., the tweet author and its number of followers, as well as the day when the tweet was posted.

The novelty of our proposed architectures is threefold:

(1) we correlate and discover new insights between the relation of *trending news topics* and *Twitter events* in both directions, i.e., *trending news topics* → *Twitter events* and *Twitter events* → *trending news topics*;
(2) we manage to create accurate models that predict the audience interest in *trending news topics* on social media and determine if they become viral;
(3) we improve the model's performance by incorporating in our document embeddings and learning models the assumption that *influencers* play an important role in the spreading as well as the day when the tweet was posted, thus making *trending news topics* to become viral.

Experimental results show that all *trending news topics* are correlated to at least one *Twitter event*, whereas the reverse is not true. Furthermore, we obtain better audience interest prediction by enhancing the dataset with metadata.

The rest of this paper is structured as follows. Section 2 presents a survey of state-of-the-art-methods for analyzing news diffusion on social media. In Section 3, we discuss the models and techniques used in our deployed solution. Section 4 presents the architecture of our system and discusses each module in detail. Section 5 showcases and discusses the obtained results. Lastly, in Section 6 we conclude and we present several new directions and improvements for the proposed solution.

2 RELATED WORK

Online media content shapes people's perceptions regarding ongoing social, political, and economical changes around them. In the literature, the relevance of the story selection correlated to a specific audience, done by editors or by algorithms, has been analyzed to find better ways to get news online and explore the relationships that exist between individuals' characteristics and their interests. The results show that the audience's interest can be determined by context-specific characteristics [34]. Furthermore, the prediction of the audience's interest in a *news topic* can help publishers to create recommendation systems for social platforms that leverage tailor-made latent features [5].

One solution for building better recommendation systems for targeting the interested audience proposes to analyse the content of posts to detect bursty topics. The analysis predicts the emergence of current events that are of interest to multiple groups of people who discuss and share the content online [1, 15, 31]. Furthermore, community-based probabilistic algorithms can be used to model the spreading of *news events* on social networks [25]. Therefore in this paper, we use event detection techniques to match *news topics* with *news events* and extract *trending news topics*. Then, we correlate the *trending news topics* with *Twitter events* using the same time frame to determine the spread of current news topics on the social media platform and determine *viral topics*.

The visibility of news on social media also depends on the actions of a diverse set of actors, e.g., the users, their friends, content publishers such as news organizations, advertisers, and algorithms [33]. Thus, we propose a new feature construction method that incorporates metadata into the features of the training set.

Deep Learning architectures have been successfully used for predicting the trend of stocks [19], financial time series [28], marketing [26], etc. Deep Learning architectures have also been used to predict information diffusion in social media. Recurrent Neural Networks (RNN) architecture is a popular deep learning-based model used to model information diffusion, obtaining promising performance. In [36], the authors explore the advantages of using an RNN-based model enhanced with reinforcement learning in order to predict information diffusion in social media. In their work, they tackle diffusion prediction both at the user level (microscopic), i.e., the next influenced user, and at the network level (macroscopic). The

proposed solution, FOREST (reinFOrced REcurrent networks with STructural context), consists of two models: the microscopic and macroscopic cascade models. The microscopic model employs a Gated Recurrent Unit architecture and a structural context extraction algorithm based on neighborhood sampling. This model is fed to the macroscopic model that also applies a reinforcement learning based cascade simulation process. For experiments, they used three datasets, each from different social media platforms. FOREST is compared with other state of the art solutions that employ Long Short-Terms Memory, attention layers, and Convolutional or Recurrent Neural Nentwork architectures. The authors use the Mean-Square Log-Transformed Error (MSLE) metric for comparison. FOREST outperforms all the baseline solutions. In comparison with our work, FOREST does not detect news topics and does not analyze from this point of view the spread of information. Also, their models ignore the timestamp information. A similar solution to [36] that, in contrast, considers the temporal information, but which still does not address audience prediction on particular topics, is proposed in [6]. Their proposed method, called CasCN (Recurrent Cascades Convolutional Networks), is also based on an RNN architecture. CasCN predicts cascades through learning the latent representation of both structural and temporal information.

To the best of our knowledge, there are no current solutions that connect news articles and social media events in order to predict the spread of certain news topics in social media. To address this shortcoming, we analyze both news articles and tweets in order to correlate the events from the two type of sources and employ Deep Learning architectures together with our feature engineered training set for an accurate audience interest prediction on particular news topics.

3 METHODOLOGY

In this section, we discuss the core components of our solution.

3.1 Term Weighting Schemes

In Information Retrieval and Text Mining, a weighting scheme is a statistical measure to evaluate how important a term is to a document in a collection or corpus. Weighting schemes are the basis for many document vectorization techniques, such as the vector space model where each feature is a word (term), and the feature's value is a term weight.

For a corpus of documents $D = \{d_1, d_2, ..., d_n\}$, where $n = ||D||$ is the total number of documents in the dataset, a document d_i is defined as a sequence of terms t_{ij}, $d_i = \{t_{i1}, t_{i2}, ..., t_{im}\}$ where m is the length of the vocabulary. The vocabulary is the set of distinct words that appear in the corpus of documents. The term frequency $TF(t_{ij}, d_i)$ of a term t_{ij} is equal to the number of co-occurrences of that term in a document d_i (Equation (1)).

$$TF(t_{ij}, d_i) = f_{t_{ij}, d_i} \tag{1}$$

The inverse-document frequency $IDF(t_{ij}, D)$ is a statistical measure of the importance of a term in a text document collection (Equation (2)), where t_{ij} is the term, D is the corpus of documents, $n = ||D||$ is the number of documents in the corpus, and n_{ij} is the number of documents where term t_{ij} appears. Terms with a low document frequency add more information than terms with a high frequency. Thus, the more frequently the term appears in the collection, the less informative the term is.

$$IDF(t_{ij}, D) = \log_2 \frac{n}{n_{ij}} \tag{2}$$

Term frequency-inverse document frequency ($TFIDF(t_{ij}, d_i, D)$) is a statistical measure used to determine the importance of a word regarding its frequency in a document relative to the entire corpus. The term importance is proportional to the number of times a word appears in the document, although it is counterbalanced by the frequency of that word in the corpus (Equation (3)).

$$TFIDF(t_{ij}, d_i, D) = TF(t_{ij}, d_i) \cdot IDF(t_{ij}, D) \tag{3}$$

The normalized term frequency-inverse document frequency $TFIDF_N(t_{ij}, d_i, D)$ (Equation (4)) uses the ℓ^2-norm (Equation (5)) to normalize the values of $TFIDF(t_{ij}, d_i, D)$, for each term t_{ij} in each document d_i, in the $[0, 1]$ interval.

$$TFIDF_N(t_{ij}, d_i, D) = \frac{TFIDF(t_{ij}, d_i, D)}{\ell^2(d_i)} \tag{4}$$

$$\ell^2(d_i) = \sqrt{\sum_{t_{ij} \in d_i} \left(TFIDF(t_{ij}, d_i, D)\right)^2} \tag{5}$$

Using the weights, we construct document-term matrices $A \in \mathbb{R}^{n \times m}$ to describe the frequency of terms that occur in the dataset. By considering this representation, rows correspond to documents and terms to columns.

3.2 Topic Modeling

Topic modeling is a statistical unsupervised machine learning method used to extract hidden latent semantic patterns within a corpus of documents. Topic modeling algorithms use either statisical models, i.e., Probabilistic Latent Semantic Indexing (PLSI) [17], generative statistical models, i.e., Latent Dirichlet allocation (LDA) [4], or matrix factorization, e.g., Non-Negative Matrix Factorization (NMF) [2], Latent Semantic Analysis [9] (LSA). Experimental results prove that NMF is the best choice for extracting topics [7].

Non-Negative Matrix Factorization. NMF is an algorithm that factorizes a matrix $A \in \mathbb{R}^{n \times m}$ into two non-negative matrices $W \in \mathbb{R}^{n \times k}$ and $H \in \mathbb{R}^{k \times m}$. In the case of topic modeling, the matrices have the following signification:

(1) A is a document-term matrix constructed using weighted term frequencies for a corpus containing n documents and a vocabulary of size m terms;
(2) W is the document-topic matrix that assigns a document membership to each topic k;
(3) H is the topic-term matrix that assigns to each topic k the importance of a term.

To determine W and H, the objective function $F(W, H)$ must be minimized by respecting the constraint that all the elements of W and H are non-negative. Equation (6) presents the objective function, where $|| \cdot ||_F$ is the Frobenius norm.

$$F(W, H) = ||A - WH||_F^2 = \sum_{i=1}^{n} \sum_{j=1}^{m} (A_{ij} - (WH)_{ij})^2 \tag{6}$$

To minimize the objective function (Equation (7)), the values of W and H are updated iteratively (with t the index of the iteration) until they stabilize (Equation (8)).

$$\min_{W \geq 0, H \geq 0} F(W, H) = \min_{W \geq 0, H \geq 0} ||A - WH||_F^2 \tag{7}$$

$$H_{ij}^{t+1} \leftarrow H_{ij}^{t} \frac{((W^t)^T A)_{ij}}{((W^t)^T W^t H^t)_{ij}}$$
$$W_{ij}^{t+1} \leftarrow W_{ij}^{t} \frac{(A(H^{t+1})^T)_{ij}}{(W^t H^{t+1}(H^{t+1})^T)_{ij}} \quad (8)$$

3.3 Information Diffusion

Information diffusion in social media studies the data propagation to find events and forecast their spreading [13]. Event detection is a subdomain of information diffusion that aims to discover real-world events from the social media [38]. We choose Mention-Anomaly-Based Event Detection (MABED) [15] as the *Twitter event* detection algorithm. MABED is a statistical event detection on social media method immune to the topic bias added by the texts unrelated to the event.

Mention-Anomaly-Based Event Detection. MABED is an efficient method for event detection that filters irrelevant content and successfully removes spam messages with no actual intent, posted around certain hours.

To identify a bursty topic and detect an event for a period of time $I = [a; b]$ and a main word t (event label) with MABED, a weight $w_{t'_q}$ is computed for each candidate word t'_q (event keywords) in the time slice i (Equation (9)). The weight is computed using the affine function $\rho_{O_{t,t'_q}}$ (Equation (10)) that corresponds to the first order auto-correlation of the time-series for N_t^i (number of tweets in the time-slice i that contain the main word t) and $N_{t'_q}^i$ (number of tweets in the time-slice i that contain the candidate word t'_q) [12].

$$w_{t'_q} = \frac{\rho_{O_{t,t'_q}} + 1}{2} \quad (9)$$

$$\rho_{O_{t,t'_q}} = \frac{\sum_{i=a+1}^{b} A_{t,t'_q}}{(b-a-1)A_t A_{t'_q}} \quad (10)$$

Where:

(1) $A_{t,t'_q} = (N_t^i - N_t^{i-1})(N_{t'_q}^i - N_{t'_q}^{i-1})$;
(2) $A_t^2 = \frac{\sum_{i=a+1}^{b}(N_t^i - N_t^{i-1})^2}{(b-a-1)}$;
(3) $A_{t'_q}^2 = \frac{\sum_{i=a+1}^{b}(N_{t'_q}^i - N_{t'_q}^{i-1})^2}{(b-a-1)}$.

3.4 Text Representation using Embeddings

Before using machine learning models for classification, prediction, or clustering, the documents must be transformed from textual data to numerical data.

Word Embedding. The Word to Vector model (Word2Vec) is a shallow neural architecture that produces a vector space for a textual dataset using the values of the neural network hidden layer [27]. There are two approaches proposed in the literature:

(1) Continuous Bag-Of-Words model (CBOW) which accounts for the textual dataset vocabulary and represents documents as a set of continuous word-multiplicity pairs. The input to the hidden layer connections is replicated by the number of context words, and the context is preserved through the use of multiple words that target a given word.
(2) Skip-gram model which represents documents as sequences of words with gaps between them. The input to the neural network is the target word, and the output layer is replicated multiple times to accommodate the chosen number of context words. Thus, this model manages to embed in the word representation its linguistic context.

The two models mirror each other while both preserve context. The CBOW model uses multiple neighbouring words to preserve the context for a target word, while the Skip-gram model uses a word to preserve the context for multiple targeted neighbouring words.

Document Embedding.

The Document to Vector (Doc2Vec) model learns continuous distributed vector representations for textual data [23]. The text dimension may vary from phrases and sentences to large documents. The model is similar to the Word2Vec model which maps words into the vector space maintaining the semantic similarities by using a given word's context. There are two approaches in the literature [23]:

(1) The Paragraph Vectors Distributed Memory (PVDM) model [23] extends the CBOW architecture by mapping each document to a vector via an additional document-to-vector matrix and concatenating this vector to the word vectors in order to predict the central word.
(2) The Paragraph Vectors Distributed Bag of Words (PVDBOW) predicts the central word using the same mechanism as PVDM model but does not preserve the word order and ignores the context.

Similarity. Using word or document embeddings together with the cosine similarity [24], the semantic similarity between two words or documents can be computed [20]. This method assumes that two embeddings have a non-zero norm and measures their orientation instead of their magnitude, as in the case of the Euclidean distance. Equation (11) presents the cosine similarity for two p-dimensional vectors x and y.

$$cos(\theta) = \frac{\sum_{i=1}^{p} x_i \cdot y_i}{\sqrt{\sum_{i=1}^{p} x_i^2} \cdot \sqrt{\sum_{i=1}^{p} y_i^2}} \quad (11)$$

3.5 Deep Learning Architectures

Artificial neural networks (ANNs) use processing units to predict an output $y \in \mathbb{R}^{n \times k}$ for an input dataset $X \in \mathbb{R}^{n \times m}$. For the given input containing feature vectors $x_i \in X$, the processing unit will try to predict an output $\hat{y}_i = \delta(\sum_{j=1}^{m}(w_{ij} \cdot x_{ij}) + b)$ using a weight vector w_{ij}, the bias b, and the activation function $\delta(\cdot)$. The activation function has different forms depending on the processing unit (Table 1).

Table 1: Activation functions

Name	Function
Sigmoid	$\delta(z) = \sigma_g(z) = \frac{1}{1+e^{-z}}$
Hyperbolic	$\delta(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
ReLU	$\delta(z) = \max(0, z)$
Softmax	$\delta(z) = \frac{e^z}{\sum_{i=1}^{m} e^{z_i}}$ $(m = dim(z))$

To accurately predict the output y, the ANN models minimize the loss or cross entropy function (Equation (12)) by adjusting the weights after a specified finite number of iterations or when the function stabilizes.

$$L(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y})) \quad (12)$$

Stochastic gradient descent (SGD) (Equation (13)) is used to update the weight vector at iteration t using the weights computed during the previous iteration. Equation (14) presents the weigths update function, where η is the global learning rate and $\alpha \in [0; 1]$ is the exponential decay factor.

$$\gamma_t = \nabla_{\boldsymbol{w}^{(t)}} L(\hat{y}, y) \tag{13}$$

$$\triangle \boldsymbol{w}^{(t)} = \alpha \triangle \boldsymbol{w}^{(t-1)} - \eta\gamma_t \tag{14}$$

ADAGRAD is a method used to improve SGD by increasing the learning rate when the $\boldsymbol{w}$ weight vectors are sparse [11]. Equation (15) presents update rule used by ADAGRAD, where $||\gamma||_2$ is the ℓ^2-norm of all previous gradients on a per-dimension basis.

$$\triangle \boldsymbol{w}^{(t)} = -\frac{\eta}{||\gamma||_2}\gamma_t \tag{15}$$

ADAGRAD has two problems :

(1) the continual decay of learning rates throughout training, and
(2) the need for a manually selected global learning rate.

ADADELTA [39] solves these two problems by using the Root Mean Square (*RMS*) to update the weights (Equation (16)).

$$\triangle \boldsymbol{w}^{(t)} = -\frac{RMS[\triangle \boldsymbol{w}]_{t-1}}{RMS[\gamma]_t}\gamma_t \tag{16}$$

Two Deep Learning Architectures used successfully in classification [21] are Multi-Layer Perceptron and Convolutional Neural Network.

Multi-Layer Perceptron. One of the basic processing units is the perceptron, which maps its input x_i to a single binary value $\hat{y}_i \in \{0, 1\}$. The perceptron can generalizes naturally to a multi-class perceptron to predict $\hat{y} = argmax_y f(x, y) \cdot \boldsymbol{w}$ by using a feature representation function $f(x, y)$ that maps each possible input/output pair to a finite-dimensional real-valued feature vector and multiplies it by a weight vector $\boldsymbol{w}$.

The Multi-Layer Perceptron (MLP) is a feed-forward ANN architecture that stacks perceptron units into three fully connected weighted directed layers:

(1) the input layer,
(2) the hidden layer, and
(3) the output layer.

The MLP becomes a Deep Feed-Forward Neural Network architecture by adding multiple hidden layers

Convolutional Neural Network. A Convolutional Neural Network (CNN) is an ANN architecture similar to the MLP, except it contains multiple hidden layers. These hidden layers consist of a series of convolutional layers that apply a filter to the activation function. They include the following types of layers: pooling, fully connected, and normalization layers. The pooling layer is used to reduce the dimensions of the data by combining the outputs of one layer into a single neuron in the next layer.

3.6 Evaluation Methods

Multi-class classification models assign a data point to one and only one non-overlapping class c_i ($i = \overline{1, k}$) [32]. To evaluate the quality of the model we can use the average accuracy (Equation (17)) where:

- TP_i (True Positive) is the number of data points correctly classified with class C_i;
- FN_i (False Negative) is the number of data points incorrectly classified with a class C_j ($j \neq i$);
- FP_i (False Positive) is the number of data points incorrectly classified with class C_i;
- TN_i (True Negative) is the number of data points correctly classified with a class C_j ($j \neq i$).

$$A = \frac{1}{k}\sum_{i=1}^{k}\frac{TP_i + TN_i}{TP_i + FN_i + FP_i + TN_i} \tag{17}$$

4 PROPOSED SOLUTION

Figure 1 presents the architecture of the proposed solution. The architecture is modular and each module is described in the following paragraphs. The code is publicly available on GitHub[1].

4.1 Data Collection and Storage Modules

To collect the News corpus we have used two APIs (Application Programming Interface): News River API[2] and NewAPI[3]. News River API returns the latest 100 news based on a given subject, e.g., politics, brexit, etc. NewAPI is a public and free API for news that contains news from many sources and can be configured to request the latest 100 news from the most popular news publisher, e.g., The New York Time, Reuters, The Washington Times, etc. Although, NewsAPI provides multiple metadata about each news article, e.g., title, description, online location, etc, the content is reduced to the first paragraph. We developed a scrapper to obtain the entire content of the article. To collect tweets, we used the Twitter API to request tweets with specific keywords and usernames. Besides the content of the tweet, we also store likes, retweets, and creation time for each tweet. The datasets are stored in a MongoDB[4] database.

4.2 Preprocessing Modules

Text preprocessing is done differently depending on the task, i.e., Topic Modeling or event detection, and type of corpus, i.e., News or Tweets. Thus, we create three preprocessed corpora:

(1) NewsTM with news articles for topic modeling,
(2) NewsED with news articles for event detection, and
(3) TwitterED with tweets for event detection.

To create the NewsTM corpus, we employ the following perprocessing steps:

(1) extract named entities to treat them as concepts and not as simple terms,
(2) extract lemmas to minimize the vocabulary and store only the base root, and
(3) remove punctuation and remove stop words because they do not add any information gain.

Both NewsED and TwitterED corpora are created using two preprocessing steps:

(1) removal of punctuation, and
(2) tokenization.

We choose this simple preprocessing pipeline to replicate the text preprocessing done originally for the MABED algorithm. The preprocessing pipeline is implemented in SpaCy[5]. The results are stored in the MongoDB database.

[1] https://github.com/cipriantruica/news_diffusion
[2] https://newsriver.io/
[3] https://newsapi.org/
[4] https://www.mongodb.com/
[5] https://spacy.io/

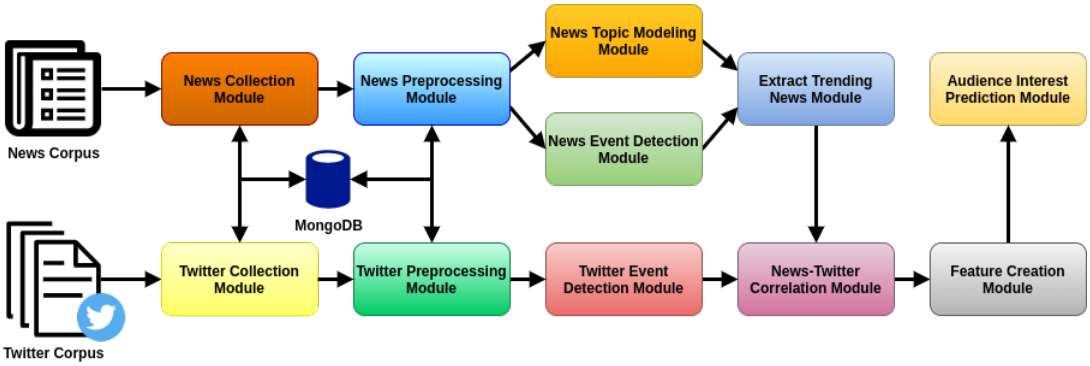


Figure 1: Architecture

4.3 Topic Modeling Module

Topic modeling is used to create an overview of the current and of interest subjects in the media. We use the NMF algorithm to extract the main topics from the NewsTM corpus. A document-term matrix is constructed from the NewsTM corpus after each document is vectorized using the $TFIDF_N$ weight. We use the Scikit-learn[6] implementations for $TFIDF_N$ and NMF.

4.4 Event Detection Modules

For both news articles and tweets, we use the MABED algorithm to detect trending topics. We enhance the NewsED and TwitterED datasets with the creation time of each record. MABED detects events defined by three characteristics:

(1) a set of main words,
(2) a set of related words, and
(3) the period of time when the topic is of interest.

We use the original implementaion of MABED[7].

4.5 Trending News Module

To correlate the news topics to the *news events*, we use the Doc2Vec model to encode the topic keywords (NewsTopic2Vec) and the *news events*' main and related terms (NewsEvent2Vec). Using the cosine similarity (Equation (11)) implemented in SpaCy, we score the match between each topic and each event. We extract the best matches for each topic, and with the help of each matching event, we determine how topics are diffused in mass media.

4.6 Correlation Module

The *news events* are correlated to the *Twitter events* to determine how current information is propagated on social media. As in the case of the *news event*, we use Doc2Vec to encode each *Twitter events*' main and related terms into one vector (TwitterEvent2Vec). We correlate the news and *Twitter events* that appear in the same time interval and extract candidate correlations. The cosine similarity implemented in SpaCy is used to find the best matches from the candidate correlations. Using this method we determine how news is propagated in social media.

[6]https://scikit-learn.org/stable/
[7]https://github.com/AdrienGuille/pyMABED

4.7 Feature Creation Module

Using the *Twitter events* detected by the Correlation Module, we extract the tweets that belong to each event. We consider that a tweet is part of an event if both of the following conditions are true:

(1) it was posted during the event's period of time, and
(2) its textual content contains at least one main word and 20% of the related words.

An event is considered of interest if there are at least 10 records associated to it. We constructed a new sub-dataset using this criteria. We encode each tweet belonging to an event using Word2Vec on the tweet's terms present in the vocabulary containing the main and related terms of that event. We use the Gensim[8] Word2Vec implementation. We then create three custom Doc2Vec embeddings for each tweet by averaging the Word2Vecs as follows:

(1) SW_Doc2Vec: only Word2Vecs found in the pre-trained model are considered in computing the document embedding;
(2) RND_Doc2Vec: random vectors with values in the range $[-1, 1]$ for terms that are not found in the pre-trained model are added before computing the document embedding;
(3) SWM_Doc2Vec: Word2Vecs found in the pre-trained model are multiplied by the word's magnitude in the context of the event before computing the document embedding.

Before creating the datasets for predictions, we also incorporate metadata into the representation of each record. The metadata vector incorporates a one-hot-encoder vector that embeds the author of each tweet, its number of followers, and the day of the week. We added these features because it was proven empirically that *i)* the number of followers influences the propagation of news in social media [16], and *ii)* the media consumption is influenced by the day of the week [3].

We also create a feature that represents the number of followers, during the period of the event, for each user. The likes and retweets classes used for predicting the events interest on social media are constructed similarly to the feature that encodes the user's number of followers. Table 2 presents the encoding for the number (#) of user followers, tweet likes, and retweets, respectively.

[8]https://radimrehurek.com/gensim/

Table 2: Features Encoding

Feature	# < 100	# ∈ [100, 1 000]	# > 1 000
followers	0	1	2
likes	0	1	2
retweets	0	1	2

4.8 Audience Interest Prediction Module

To predict the audience interest in a news topic, we analyze the retweets and likes received by the tweets that discuss the news articles belonging to that topic. We use two Deep Neural Networks architectures for our prediction module:

(1) a Multi Layer Perceptron (Figure 2), and
(2) a Convolutional Deep Learning architecture (Figure 3).

The first architecture uses only perceptron units that can easily generalize a wide variety of problems. The second architecture uses a convolution layer and a max pooling layer. We use accuracy to evaluate the networks' performance and classification's quality.

4.9 Design choices

We decided to fetch the latest tweets and news every 2 hours. Thus, we leave a large enough time window to manage to collect a representative number of new tweets and news articles. The algorithms are executed, from checkpoints or from scratch, after each dataset update, and the models are replaced with the new ones as soon as they finish. Thus, multiple instances of the same algorithm may run at the same time on different datasets. Also, we choose to use NMF instead of LDA [4] as it provides similar results on both small and large length texts in less time [35].

For training the word embeddings and then vectorizing the textual data to extract Doc2Vec, we choose a pretrained word2vec on the Google News corpus[9]. This dataset contains 3 million 300-dimension English word vectors. We made this design choice because this dataset is larger than the datasets we collected and manages to provide better word representations on which to construct our Doc2Vec embeddings. In contrast, the PVDM and PVDBOW models are not good for our study because they will not find good document representations since they can be trained by us only on the collected datasets. Thus, these models do not manage to generalize the document representation.

To alleviate the need to train the neural models each time the datasets are updated, we use checkpoints to continue the training as new data is added in real time. Furthermore, the most demanding networks are trained in less than 7 minutes. Thus, the models can be build again if all data is new.

5 EXPERIMENTAL RESULTS

We apply our method on a real-world data, aiming to eventually predict user interest.

5.1 Datasets

We have collected 261 052 news articles, and 80 569 tweets for a period of 5 months. Both datasets contain records from different news venues and are all written in English. For the Twitter corpus, we also collected statistics for both tweets, i.e., likes and retweets, and users, i.e., number of followers, the friends count, and the retweets count.

[9]https://code.google.com/archive/p/word2vec/

5.2 News Topics

The first set of experiments uses the NewsTM corpus together with NMF algorithm to extract the most relevant 100 topics from all the news articles. This process takes 19.01 minutes. Table 3 presents a subset of 10 *news topics* (NT) extracted for the entire period. These news topics are used to showcase the correlation with the corresponding *Twitter events*.

Table 3: News topics

#NT	Keywords
1	party election vote seat poll voter conservative win european brexit
2	tariff import billion chinese good impose 25 consumer product percent
3	company business market industry customer service growth product year technology
4	trade deal market war global economy talk agreement tension china
5	huawei company google ban smartphone android chinese network security technology
6	iran iranian tehran sanction nuclear drone tension deal gulf tanker
7	israel gaza israeli palestinian hamas rocket militant palestinians jerusalem netanyahu
8	japan abe japanese emperor tokyo naruhito shinzo visit imperial meet
9	impeachment pelosi democrats impeach nancy inquiry speaker house proceeding congress
10	derby horse kentucky race win belmont maximum winner security racing

5.3 News Events

For our experiments, we extract the top 1 000 *news events* from the NewsED corpus, using the MABED algorithm. We use a time frame of 60 minutes. The extraction of *news events* takes 17.08 hours: 177.41 seconds to load the corpus, 1.3 hours to partition the news into time-slices, and 15.73 hours to extract events. Table 4 presents a subset of the *news events* (NE) detected by MABED. These *news events* are some of the *trending news articles* determined during the correlation with the *news topics*.

5.4 Twitter Events

Using MABED, we identified the top 5 000 events on the TwitterED corpus collected for a period of 5 months that have at least 10 tweets associated to it. We use a time window of 30 minutes. The extraction of *Twitter events* takes 11.74 hours: 1.61 seconds to load the corpus, 70.42 seconds to partition the news into time-slices, and 11.72 hours to extract events. Table 5 presents some of the detected *Twitter events* (TE) that we use to exemplify the correlation between *trending news topics* and *Twitter events*.

5.5 Correlation Results

We use cosine similarity to correlate the *news topics* (NT) with *news event* (NE) to extract *trending news topics*. Then, we correlate *trending news topics* with *Twitter events* (TE), i.e., *trending news topics* → *Twitter events* . First, we correlate each *news topics* with each *news event* and extract the ones with the highest similarity.

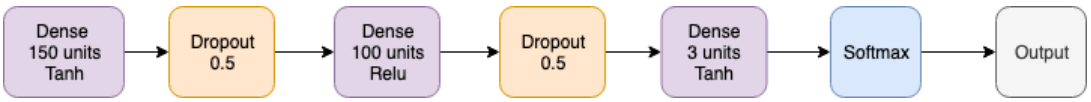


Figure 2: The MLP Network Architecture

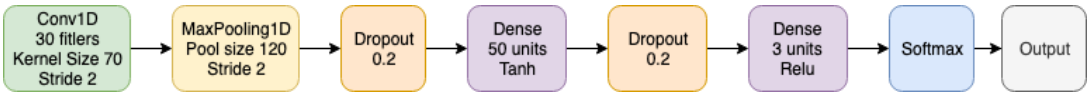


Figure 3: The CNN Network Architecture

Table 4: News events

#NE	Start Date	Start Date	Label	Keywords
1	2019-05-11 03:05:40	2019-05-26 13:05:40	politics	political european eu current election vote campaign voters
2	2019-05-11 03:05:40	2019-05-26 13:05:40	reached	current developing socialnews future 2019 group march company
3	2019-05-11 05:05:40	2019-05-26 13:05:40	plans	current prime group future business company vote european
4	2019-05-11 05:05:40	2019-05-26 13:05:40	comes	current future business part week north company american political
5	2019-05-11 05:05:40	2019-05-26 13:05:40	mobile	huawei current developing socialnews future chinese gopi adusumilli
6	2019-05-05 23:05:40	2019-05-26 13:05:40	threats	iran nuclear washington waters foreign american
7	2019-05-03 19:05:40	2019-05-11 11:05:40	conflict	military gaza israeli killed group hamas islamic political
8	2019-05-21 09:05:40	2019-05-26 13:05:40	japanese	japan abe tokyo north prime huawei tariffs american visit donald
9	2019-05-09 17:05:40	2019-05-22 01:05:40	familiar	white democrats committee administration congress
10	2019-05-02 19:05:40	2019-05-07 19:05:40	bob	derby security win mueller kentucky times

Table 5: Twitter events

#TE	Start Date	End Date	Label	Keywords
1	2019-04-29 20:01:30	2019-07-10 02:01:30	conservative	party theresa brexit leader mps prime minister leadership
2	2019-05-06 02:01:30	2019-06-27 14:01:30	fresh goods	tariffs threaten china trade good escalation import stock
3	2019-05-01 02:01:30	2019-08-09 02:01:30	improving	retail conservative taking actions people life growth online
4	2019-05-08 14:01:30	2019-07-31 08:01:30	war brand	big tax trade us-china markets billion conservative companies
5	2019-05-14 20:01:30	2019-07-12 14:01:30	networks	trump declare national emergency protect computer foreign web
6	2019-04-17 08:01:30	2019-07-25 02:01:30	muslim	aide biden hill bombshell betting handle joe contempt capitol
7	2019-04-19 20:01:30	2019-07-26 08:01:30	impeachment	democrats trump mueller pelosi testimony politically voted
8	2019-04-06 02:01:30	2019-05-22 08:01:30	woods tale	tiger victory roll lawsuit nationals back-to-back grand horse
9	2019-05-09 20:01:30	2019-05-21 02:01:30	senate alabam	passed effectively abortion ban bill committee
10	2019-05-07 08:01:30	2019-08-12 20:01:30	english fans	football manchester club everton fantasy clubs playing

The pairs < *news topics*, *news event* > are the *trending news topics*. We extracted 83 *trending news topics* that have a similarity of over 0.7. Then, for each *trending news topics*, we extract the *Twitter events* in the same time window with the highest Doc2Vec similarity. Given the start date (S_{TT}) of a *trending news topics* (TT), the constraint of the *Twitter events* (TE) start date is $S_{TE} \in [S_{NE}; S_{NE} + 5day]$. We choose this start interval because a *Twitter event* can appear on social media as soon as the news appears in the mass media, but it can also be some delay between the appearance time on the two platforms. The end date is not significant as a *Twitter event* can be prolonged even though the mass media stops releasing new content. We determined a total of 421 < *trending news topics*, *Twitter events* > pairs that respect the time constraint and have a similarity of over 0.65. This process takes 31.2 minutes: 17.41 minutes to extract *trending news topics* and 13.79 minutes to determine the < *trending news topics*, *Twitter events* > pairs.

Table 6 showcases a subset of the correlation between the selected *news topics*, *news events*, and *Twitter events*, where the topic and event numbers can be found in Tables 3, 4, and 5. As we extract 1 000 *news events*, in Table 6 the *news events* also represent *trending news topics*. The presented correlations also include the best and the worst similarities.

Table 6: Correlation between topics and events

#NT	#NE	#TE	Sim NT NE	Sim NE TE
1	1	1	0.87	0.88
2	2	2	0.73	0.79
3	3	3	0.86	0.89
4	4	4	0.78	0.85
5	5	5	0.77	0.78
6	6	6	0.84	0.75
7	7	7	0.90	0.79
8	8	8	0.78	0.77
9	9	9	0.82	0.81
10	10	10	0.77	0.69

The matching between the pair < *news topics*, *news events* > and *Twitter events* shows that the news articles that appear in

Table 7: Unrelated Twitter Events

#TE	Start Date	End Date	Label	Keywords
1	2019-03-08 08:01:30	2019-07-12 14:01:30	cartoon	matt cartoonist telegraph side bobs cartoons
2	2019-04-12 08:01:30	2019-05-03 14:01:30	social media	whatsapp facebook videos mark zuckerberg user
3	2014-10-30 08:01:30	2019-05-21 02:01:30	game of thrones	spoilers season episode missed review sunday
4	2019-08-02 20:01:30	2019-08-14 02:01:30	sleep	coffee news lovers tea studying perfect ashes
5	2019-04-13 14:01:30	2019-07-20 02:01:30	rice	delicious perfectly sandwiches fried dish cheeses

mass media are discussed in detail during their publication on social media. Although not all *news topics* are directly related to a *Twitter events*, e.g., NT #9 discussed Trump's impeachment, and TE #9 discusses the abortion law passed in Alabama, the correlation between *news topics* and *news events* is high. Thus, these topics are considered *trending news topics*. Furthermore, some *trending news topics* match generalized *Twitter events*, e.g., NT #10 is related to the Kentucky derby, where the winning horse named Maximum Security was disqualified, while TE #10 is related to Manchester football clubs, but both talk about sports. Other matches are quite specific, e.g., NT #1 and TE #1, which both are related to European politics and Brexit.

In conclusion, a similarity over 0.77 between *news topics* and *news events* shows that the topics and events are consistent, while a similarity over 0.69 between *news events* and *Twitter events* denotes a generalization tendency of events for *trending news topics*. Furthermore, we found that all the *trending news topics* have correlations with at least one *Twitter event* and that some *trending news topics* are correlated to the same *Twitter events*.

We also analyzed the reverse correlation, i.e., *Twitter events* → *trending news topics*, by applying the same steps as for *trending news topics* → *Twitter events* but in reverse. We observe that the set given by the correlation *Twitter events* → *trending news topics* is the same as the one given by *trending news topics* → *Twitter events*.

However, as Twitter is a social media platform for generic discussion, some events discovered by MABED within the tweets collected are not related to the current *news events*. Thus, we can conclude that some thread discussions present generic topics. They spam for longer periods of time and the keywords are more generic. These events are related to common discussions regarding food, social media in general, television shows, etc. Table 7 presents some of the tweeter events that are not related to any of the news articles topics and do not match any *trending news topics* using our matching condition.

5.6 Audience Interest Prediction

Using the correlation between *trending news topics* and *Twitter events* we want to predict the interest of twitter users in *news topics* on social media using metadata, i.e., the user and its number of followers, and determine if a *news topic* becomes viral. We employ the MLP and CNN architectures to predict the audience interest in the *trending news topic*. The interest is given by both the number of retweets and likes received by the *Twitter Events* correlated to the *trending news topics*. We train each network on a machine with a 3.5GHz quad-core processor and 16GB RAM, using the Keras[10] library with TensorFlow[11] as backend.

Each network is trained until it converges, using an Early Stopping mechanism that checks if there are any changes in the loss function from one epoch to the next. We used the optimizers SGD and ADADELTA and different leaning rates (lr). After hyperparameter tuning and cross validation, we obtain the following configurations which have the best results:

- MLP 1 uses the MLP architecture with the SGD optimizer and a $lr = 0.5$,
- MLP 2 uses the MLP architecture with the ADADELTA optimizer and a $lr = 2$,
- CNN 1 uses the CNN architecture with the SGD optimizer and a $lr = 0.5$, and
- CNN 2 uses the CNN architecture with the ADADELTA optimizer and a $lr = 2$.

The input of the networks contains the document embeddings (Doc2Vec) for each tweet belonging to a *Twitter event* determined by the correlation < *Twitter events*, *trending news topics* >. Thus, as some tweets can belong to multiple events, the size of the Twitter dataset increases. The label for a tweet is determined by the number of likes and retweets, respectively, as presented in Table 2. We create 8 datasets using the custom document embeddings and the metadata vector, as presented in Section 4 as follows:

- A1 uses the SW_Doc2Vec model;
- A2 uses the SW_Doc2Vec model concatenated with the metadata vector;
- B1 uses the RND_Doc2Vec model;
- B2 uses the RND_Doc2Vec model concatenated with the metadata vector;
- C1 uses the SWM_Doc2Vec model;
- C2 uses the SWM_Doc2Vec model concatenated with the metadata vector;
- D1 uses the SW_Doc2Vec model;
- D2 uses the SW_Doc2Vec model concatenated with the metadata vector and the tweet's author number of followers.

We set the document embedding size to 300. The metadata vector has a size 8 which includes an one-hot-encoding vector for the tweets' author, i.e., the *influencer* and the its number of followers, of length 7 and one element for the day of the week. Thus, we include two behaviour factors in our embedding:

(1) a temporal one given by the day which incorporate user behaviour patterns, i.e., the patterns users have to post online;
(2) a global one given by the author metadata which integrates the impact of *influencers* on the virality of *trending news topics*.

For each experiment, the behavior of the network is described by the reduction slope of the loss function and the increasing slope of the accuracy function over the training dataset. Table 8 presents the measured accuracy results over our *validation* sets on predicting *likes*, while Table 9 presents the accuracy for predicting *retweets*. We define accuracy in terms of correct

[10] https://www.keras.io/
[11] https://www.tensorflow.org/

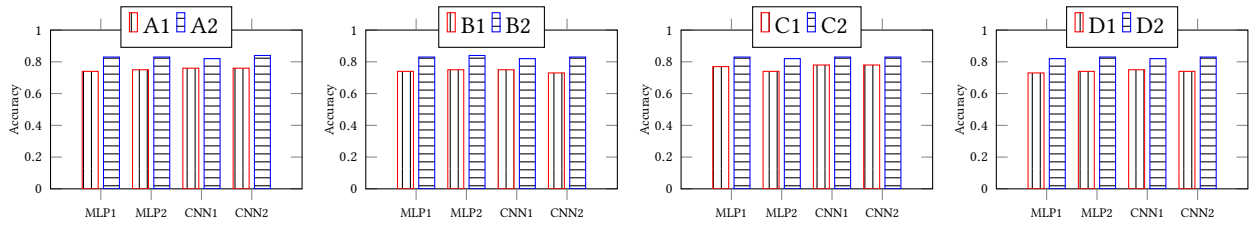


Figure 4: Likes accuracy comparison: without metadata (A1, B1, C1, and D1) vs. with metadata (A2, B2, C2, and D2)

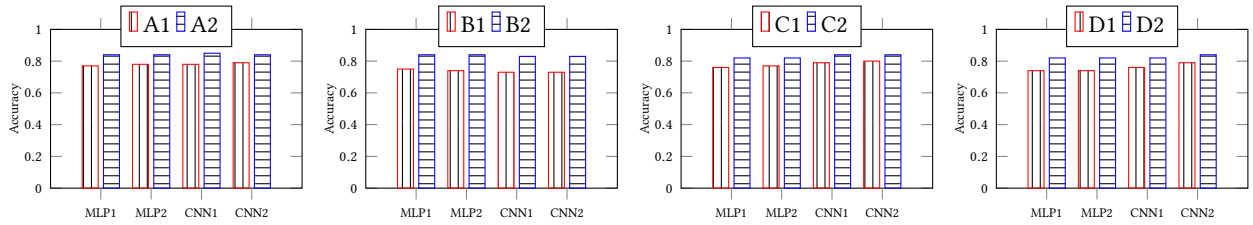


Figure 5: Retweets accuracy comparison: without metadata (A1, B1, C1, and D1) vs. with metadata (A2, B2, C2, and D2)

prediction in one of the 3 encoded classes for likes and retweets. Note that our accuracy results also reflect the error rate, since ErrorRate = 1 − Accuracy. As our results show, we successfully achieve low error rates by virtue of utilizing terms extracted using MABED and NMF; this success is based on the fact that similar topics/events have high similarity scores, while dissimilar ones have low similarity scores.

Table 8: Likes accuracy of correlated results

Dataset	MLP 1	MLP 2	CNN 1	CNN 2
A1	0.74	0.75	0.76	0.76
A2	0.83	0.83	0.82	0.84
B1	0.74	0.75	0.75	0.73
B2	0.83	0.84	0.82	0.83
C1	0.77	0.74	0.78	0.78
C2	0.83	0.82	0.83	0.83
D1	0.73	0.74	0.75	0.74
D2	0.82	0.83	0.82	0.83

Table 9: Retweets accuracy of correlated results

Dataset	MLP 1	MLP 2	CNN 1	CNN 2
A1	0.77	0.78	0.78	0.79
A2	0.84	0.84	0.85	0.84
B1	0.75	0.74	0.73	0.73
B2	0.84	0.84	0.83	0.83
C1	0.76	0.77	0.79	0.80
C2	0.82	0.82	0.84	0.84
D1	0.74	0.74	0.76	0.79
D2	0.82	0.82	0.82	0.84

We observe that both networks reach good results in terms of predicting the audience interest in specific *news topics* on social media, ranging from 0.73 to 0.85 accuracy, and accurately predict whether the *news topic* becomes viral. Furthermore, both architectures achieve similar accuracy scores regardless of the optimizer.

We conclude that these results are highly impacted by the MABED algorithm because by detecting the most trending topics and identifying the event for a tweet afterward, it basically extracts required features to predict a range of likes and retweets.

Interestingly, both architectures reach a prediction score of 0.75 after only a few epochs are finished. After these epochs, the learning process is slow and not very stable, exhibiting fluctuations between 0.78 and 0.90.

The high results are influenced by the metadata used to enhance the training corpus. We observe that the accuracy improves over the baseline with more 0.05 in some cases. This proves that the *influencer* role in spreading the news as well as the behaviour patterns of posting depending on the day have a huge impact on predicting the virality of a *trending news topic*.

The popularity of a person inside a group determines the spread of its messages, thus proving that the *influencers* assumption also holds for determining *trending news topics* on social media. In conclusion, a user that has many followers or tweets with many likes and retweets will maintain the ascending trend for future tweets with high probability. Also, we conclude that using the metadata vector improves the accuracy of prediction for all our experiments (Figures 4 and 5).

5.7 Scalability

To evaluate scalability, we extract 500, 2 500, and 5 000 *Twitter events*. We determine and encode using Doc2Vec the tweets for each event and train the networks. Table 10 presents the runtime evaluation for the networks using a batch size of 5 000 and 500 epochs. Early Stopping mechanism is used to stop the training. All the experiments are performed using the CPU. The training process for one epoch takes on average 1 second and 13/14 milliseconds for the MLP architectures (Figure 6), while the CNN architectures have a linear time increase from 1 second 71 millisecond to 6 seconds 83 milliseconds w.r.t. the number of events and the Doc2Vec size (Figure 7). This extra time is added by the complexity of the convolution layer and the number of kernel filters applied to the input vector. The CNN architectures maintain a much lower number of epochs than the MLP ones, regardless of the number of events or the Doc2Vec size. The difference between optimizers is not obvious at the batch level as

time performance and accuracy remains the same w.r.t. number of events and Doc2Vec size. We observe a difference in the number of batches, as the ADADELTA optimizer, on average, requires more batches until it converges than the SGD optimizer. We note that these results may vary, depending on the hardware used.

Table 10: Runtime evaluation

No. Twitter Events	Doc2Vec Size	Network	No. Epochs	Milliseconds Epoch	Runtime (Seconds)
500	300	MLP1	113	1013	119.51
		MLP2	119	1014	121.87
		CNN1	6	1071	6.67
		CNN2	7	1073	7.75
	308	MLP1	143	1013	154.53
		MLP2	162	1014	177.37
		CNN1	6	1073	7.16
		CNN2	8	1074	8.97
2 500	300	MLP1	316	1013	331.15
		MLP2	363	1014	381.52
		CNN1	6	3078	25.08
		CNN2	7	3079	26.95
	308	MLP1	319	1013	337.27
		MLP2	375	1014	392.24
		CNN1	6	4081	28.09
		CNN2	12	4082	52.69
5 000	300	MLP1	289	1013	334.76
		MLP2	305	1014	348.27
		CNN1	6	5097	31.85
		CNN2	7	5098	37.41
	308	MLP1	328	1013	351.87
		MLP2	368	1014	379.09
		CNN1	6	6081	38.49
		CNN2	14	6083	87.13

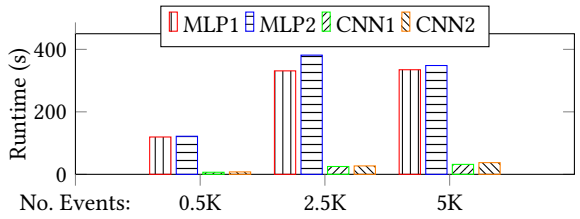


Figure 6: Performance time for 300-dimensions Doc2Vec

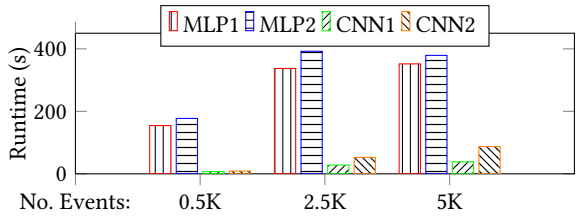


Figure 7: Performance time for 308-dimensions Doc2Vec

5.8 Discussion

Correlation between Trending News Topics and Twitter Events. To determine the *trending news topics*, we correlated the *news topics* with *news events*. Then, we correlated the *trending news topics* with the *Twitter events* that have a high cosine similarity and are in the same time window. We observe that for the same *trending news topics*, we can match multiple *Twitter events* using the imposed constraints. Thus, we can conclude that the events generated by posts on social media about *trending news topics* are intertwined. Therefore, we can conclude that some important *news topics* appear in multiple discussion and that users make connections between them in their discussion. Furthermore, all the *trending news topics* have at least one matching *Twitter event*.

We also applied the reverse correlation, by matching *Twitter events* to *trending news topics*. We discovered that we obtain the same set of pair for *Twitter events* → *trending news topics* as for *trending news topics* → *Twitter events*. During this analysis, we also discovered that multiple *Twitter events* have no correlated *trending news topics*. We attribute these results to the fact that Twitter is a generalized discussion forum. Thus, users do not address in their post only current *news events* but also discuss other events that impact their life but do not appear in mass media.

Audience interest prediction in Trending News Topics. We predict the audience interest in *trending news topics* by constructing two Deep Learning models to determine if they become viral. Thus, a *trending news topic* has a high probability of becoming viral if the posts that belong to the *Twitter event* correlated with it has a high number of likes and retweets.

For our experiments, we embedded the tweets by combining document embeddings with a metadata vector that contains the user's tweets and its number of followers as well as the day of the week. This enhanced tweet embedding feature adds two new dimension within the learning model: a temporal one given by the day and a global one given by the user metadata. The experimental results show that the two models determine with high accuracy if a *trending news topic* becomes viral for both the datasets constructed using only documents embeddings and the ones that also use the metadata vector.

Through the use of the metadata vector, we incorporate in our training set two assumptions. The first assumption is a global one and introduces the concept of *influencers* (users with a high number of followers) in the learning model. The *influencers* have a huge role in spreading the information and making *trending news topics* to become viral. The second assumption is a temporal one which introduces the user behavior based on the day of the week, as behaviour patterns can change from one day to another. Using the assumption, the neural networks learn to determine patterns in the way users post on social media and integrate them in the final prediction model. The experimental results prove that this assumption stands for our use case. Both Deep Learning models achieve an improved accuracy when the metadata vector is concatenated to the document embedding. Thus, we can conclude that the *influencers* metadata manages to create models that better generalize while improving the overall performance.

Fake news mitigation considerations. The spread of misinformation in social media has the effect of polarizing opinions and misleading readers by presenting alleged, imaginary facts about social, economic, and political subjects of interest [16]. Our method manages to predict the virality of news content and the interest the public at large has in different news topics. These findings can prove useful in designing new mitigation algorithms that take into account both textual content and network metadata. Thus, we consider that the presented system manages to determine what topics are important and can be a starting point to develop new strategies for network immunization in the fight against misinformation.

6 CONCLUSIONS

In this work, we introduce and deploy an architecture for predicting whether a *trending news topic* becomes viral on social media. We employ NMF topic modeling to extract *news topics* and MABED event detection to extract *news events* and *Twitter events*. We correlate *news topics* to *news events* by document embedding cosine similarity to extract *trending news topics*, match *trending news topics* to *Twitter events* in the same time window, and thereby extract likes, retweets, and user followers. We propose a new metadata-based document embedding for tweets associated to an event that encodes *influencers* information. Using the new document embeddings, we prepare a training corpus to predict audience interest in a *trending news topic*, using two Deep Learning architectures.

The similarity between *news topics* and *news events* in our data is over 0.77, while that between *trending news topics* and *Twitter events* is over 0.69. Our results show that the *trending news topics* and the *news events* are consistent, while the proposed architectures predict audience interest in a *news topic*, with accuracy over 0.82 in the metadata enhanced dataset.

In the future, we plan to use other matching techniques, e.g., Minimum Cost Flow, to correlate *news topics*, *news events*, and *Twitter events*, and use topic modeling [7] to see if we can extract more coherent topics from news. We also plan to employ word, sentence, and document level transformers (BERT [10], XLNet [37], ALBERT [22], ELECTRA [8]) as embeddings to take advantage of contextual information extracted using these models. Further, our solution can be included in larger solutions for fake news mitigation and misinformation immunization strategies for social media.

This work was done in collaboration with RoNews, a start-up offering news media content, which was interested in detecting article topics and verifying content veracity. Hootsuite and Sprinklr have also presented interest in this solution.

REFERENCES

[1] Loulwah AlSumait, Daniel Barbará, and Carlotta Domeniconi. 2008. On-line lda: Adaptive topic models for mining text streams with applications to topic detection and tracking. In *IEEE International Conference on Data Mining*. 3–12.

[2] Sanjeev Arora, Rong Ge, and Ankur Moitra. 2012. Learning Topic Models – Going beyond SVD. In *Annual Symposium on Foundations of Computer Science*. 1–10.

[3] Frank Bentley, Katie Quehl, Jordan Wirfs-Brock, and Melissa Bica. 2019. Understanding Online News Behaviors. In *ACM Conference on Human Factors in Computing Systems*. 1–11.

[4] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of Machine Learning Research* 3 (2003), 993–1022.

[5] Chung-Chi Chen, Hen-Hsen Huang, and Hsin-Hsi Chen. 2019. Next cashtag prediction on social trading platforms with auxiliary tasks. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. 525–527.

[6] Xueqin Chen, Fan Zhou, Kunpeng Zhang, Goce Trajcevski, Ting Zhong, and Fengli Zhang. 2019. Information diffusion prediction via recurrent cascades convolution. In *IEEE International Conference on Data Engineering*. IEEE, 770–781.

[7] Yong Chen, Hui Zhang, Rui Liu, Zhiwen Ye, and Jianying Lin. 2019. Experimental explorations on short text topic mining between LDA and NMF based Schemes. *Knowledge-Based Systems* 163 (2019), 1–13.

[8] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *International Conference on Learning Representations*. 1–17.

[9] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 41, 6 (1990), 391–407.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics*. 4171–4186.

[11] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research* 12, 61 (2011), 2121–2159.

[12] Orhan Erdem, Elvan Ceyhan, and Yusuf Varli. 2014. A new correlation coefficient for bivariate time-series data. *Physica A: Statistical Mechanics and its Applications* 414 (2014), 274–284.

[13] Ahmad Foroozani and Morteza Ebrahimi. 2019. Anomalous information diffusion in social networks: Twitter and Digg. *Expert Systems with Applications* 134 (2019), 249–266.

[14] Guiyuan Fu, Feier Chen, Jianguo Liu, and Jingti Han. 2019. Analysis of competitive information diffusion in a group-based population over social networks. *Physica A: Statistical Mechanics and its Applications* 525 (2019), 409–419.

[15] Adrien Guille and Cécile Favre. 2014. Mention-anomaly-based event detection and tracking in twitter. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. 375–382.

[16] Imane Hafnaoui, Gabriela Nicolescu, and Giovanni Beltrame. 2019. Timing Information Propagation in Interactive Networks. *Scientific Reports* 9, 1 (2019).

[17] Thomas Hofmann. 2000. Learning the Similarity of Documents: An Information-Geometric Approach to Document Retrieval and Categorization. In *Advances in Neural Information Processing Systems*. 914–920.

[18] Maryam Hosseini-Pozveh, Kamran Zamanifar, and Ahmad Reza Naghsh-Nilchi. 2019. Assessing information diffusion models for influence maximization in signed social networks. *Expert Systems with Applications* 119 (2019), 476–490.

[19] Ziniu Hu, Weiqing Liu, Jiang Bian, Xuanzhe Liu, and Tie-Yan Liu. 2018. Listening to Chaotic Whispers. In *ACM International Conference on Web Search and Data Mining*. 261–269.

[20] Tom Kenter and Maarten de Rijke. 2015. Short Text Similarity with Word Embeddings. In *ACM International on Conference on Information and Knowledge Management*. 1411–1420.

[21] Jaeyoung Kim, Sion Jang, Eunjeong Park, and Sungchul Choi. 2020. Text classification using capsules. *Neurocomputing* 376 (2020), 214–221.

[22] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *International Conference on Learning Representations*. 1–17.

[23] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International Conference on Machine Learning*. 1188–1196.

[24] Omer Levy, Yoav Goldberg, and Ido Dagan. 2015. Improving Distributional Similarity with Lessons Learned from Word Embeddings. *Transactions of the Association for Computational Linguistics* 3 (2015), 211–225.

[25] Xiaoyan Lu and Boleslaw K. Szymanski. 2018. Scalable Prediction of Global Online Media News Virality. *IEEE Transactions on Computational Social Systems* 5, 3 (2018), 858–870.

[26] Luis Miguel Matos, Paulo Cortez, Rui Mendes, and Antoine Moreau. 2019. Using Deep Learning for Mobile Marketing User Conversion Prediction. In *International Joint Conference on Neural Networks*. 1–8.

[27] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In *International Conference on Learning Representations*. 1–12.

[28] Ben Moews, J. Michael Herrmann, and Gbenga Ibikunle. 2019. Lagged correlation-based deep learning for directional trend change prediction in financial time series. *Expert Systems with Applications* 120 (2019), 197–206.

[29] Reza Motamedi, Soheil Jamshidi, Reza Rejaie, and Walter Willinger. 2019. Examining the evolution of the Twitter elite network. *Social Network Analysis and Mining* 10, 1 (2019), 1–18.

[30] Emilio Serrano, Carlos A. Iglesias, and Mercedes Garijo. 2015. A Survey of Twitter Rumor Spreading Simulations. In *Computational Collective Intelligence*. 113–122.

[31] David A. Shamma, Lyndon Kennedy, and Elizabeth F. Churchill. 2011. Peaks and Persistence: Modeling the Shape of Microblog Conversations. In *ACM Conference on Computer Supported Cooperative Work*. 355–358.

[32] Marina Sokolova and Guy Lapalme. 2009. A systematic analysis of performance measures for classification tasks. *Information Processing & Management* 45, 4 (2009), 427–437.

[33] Kjerstin Thorson, Kelley Cotter, Mel Medeiros, and Chankyung Pak. 2019. Algorithmic inference, political interest, and exposure to news and politics on Facebook. *Information, Communication & Society* (2019), 1–18.

[34] Neil Thurman, Judith Moeller, Natali Helberger, and Damian Trilling. 2018. My Friends, Editors, Algorithms, and I. *Digital Journalism* 7, 4 (2018), 447–469.

[35] Ciprian-Octavian Truică, Florin Radulescu, and Alexandru Boicea. 2016. Comparing different term weighting schemas for Topic Modeling. In *IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 307–310.

[36] Cheng Yang, Jian Tang, Maosong Sun, Ganqu Cui, and Zhiyuan Liu. 2019. Multi-scale Information Diffusion Prediction with Reinforced Recurrent Networks.. In *IJCAI*. 4033–4039.

[37] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russlan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems*. 5753–5763.

[38] Zhenguo Yang, Qing Li, Liu Wenyin, and Jianming Lv. 2019. Shared Multi-view Data Representation for Multi-domain Event Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 5 (2019), 1243–1256.

[39] Matthew D. Zeiler. 2012. Adadelta: an adaptive learning rate method. *CoRR* abs/1212.5701 (2012), 1–6.

Industrial Paper

AutoDBaaS: Autonomous Database as a Service for managing backing services*

†

Mayank Tiwary
University of British Columbia
Vancouver, Canada
mayank09@cs.ubc.ca

Pritish Mishra
University of Toronto
Toronto, Canada
pritish@cs.toronto.edu

Shashank Mohan Jain
SAP Labs Bangalore
Bangalore, India
shashank.jain01@sap.com

Kshira Sagar Sahoo
Department of CSE, SRM University AP Amravati
Mangalagiri, India
kshirasagar12@gmail.com

ABSTRACT

This work introduces and aim to overcome the potential challenges while deploying automated tuning of relational database as a service for a Platform as a Service (PaaS) provider. Some of the major challenges identified in this work include (i) automated detection of performance throttling (figure out when the performance of the system is affected due to incorrect configurations of knobs) of a database and identify potential points where a database requires a tuning, (ii) scalability and accuracy of tuning service and (iii) applying the recommendations obtained from tuning services wherein applying an obtained recommendation might require a database restart.

In this work, we present a generic tuning service architecture for PaaS providers. To deal with the above challenges, we introduce performance throttling engine which is responsible to detect potential points when a relational database actually needs a knob tuning, which helps in increasing the scalability and accuracy of the tuner deployments (responsible for tuning production landscapes). This work also proposes approaches that facilitate efficiently applying the recommendations without causing much disruption in Quality of Service (QoS) of the underlying database system. Lastly, the results are obtained by evaluation of the proposed methods and modules on multiple cloud native provisioners against various set of metrics.

1 INTRODUCTION

The PaaS customers do not have access to tune the configuration knobs of database/backing services as the service configurations are often abstracted. Tuning of the offered data services often requires DBAs to pitch in, observe/monitor and then, tune the service-instances. This often adds more complexity, as PaaS providers needs to have a DBA for each customer group, where each service offered has tens to hundreds of knobs to be tuned. In literature, there exists a set of various auto-tuners [1], [2] and [3] that aim to automate the tasks of a DBA. These tools are not holistic in nature and are limited to specific classes of parameters.

This work introduces challenges and requirements for introducing generic tuner as a service (AutoDBaaS), which can tune the configuration knobs of the relational data services as per requirement and thus, in-turn reduce the performance dependency on a DBA. In this work we evaluate already existing tuners and try to see how they can be used to tune live production systems. In literature there exists multiple style of tuners, broadly classified as search based [16] and learning based. This work specifically considers learning based tuners (as they can easily tune multiple types of databases and more suitable for PaaS service providers): bayesian optimization (BO) style tuners (like Ottertune [4]) and reinforcement learning (RL) tuners [18] and [17]. We discuss the pros and cons of both RL and BO style tuners for tuning live production systems in coming sections. Potentially, the challenges that drive the design and deployment of a tuning service as per PaaS architecture have been identified as follows:

- Scalability of Tuners
- High Quality Samples
- Metrics for Tuner Evaluation

Scalability of Tuners. As per the architecture of a BO style, it uses previously observed workloads to train a Gaussian Process (GP) regression (or a surrogate model), which recommends a new set of configs. The workload in Ottertune (any large scale machine learning tuner which tries to leverage previous experiences) is a collection of different knob values, obtained with respect to observed database metrics. The workload should contain enough data, where sufficient metric variations are observed across different variations in values of knobs. Or in another sense, a BO stlye tuner like Ottertune needs high volume of high quality samples. With the high volume samples, the Ottertune's workload size increases and causes a GPR training to take a time of around 100 to 120 seconds. Then if the underlying services, asks for recommendation with a high frequency of 5 mins (a typical monitoring time for a transactional data), one Ottertune deployment can be bound to a maximum of 3 to 4 service instances. This can also be inferred as a cost for a BO style tuners - 'recommendation-cost' to service-provider. In this aspect the RL style tuners do pretty well, as they do not need high volume of high quality samples. However, the pros and cons of BO style tuners and RL style tuners are discussed in comming sections.

High Quality Samples. Both RL and BO style tuners need to capture delta metrics (after execution of a workload) from the underlying database to be tuned. The quality of the captured metrics (or the quality of the samples) depends solely on the workload executed on the database. For example - when a client

*Produces the permission block, and copyright information
†The full version of the author's guide is available as `acmart.pdf` document

executes TPCC queries to a database, continuously for 10 minutes with 3000 requests per second, will generate a high quality sample. However, in production systems, the throughput of the executing workload is often low (or for most of the time, the production database does not needs a tuning), which cannot produce enough variations in the delta metrics (or often a low quality sample). In production systems, the spikes in throughput graph is seen at specific time-intervals only. In many cases, it is observed that even if there is high throughput, only a certain set of metrics show good variations and rest do not [4]. The quality of samples highly impacts the performance of both RL and BO style tuners. And, in production systems capturing high quality samples is very difficult (or atleast there exists no such way to do so in literature). In a nutshell, the main problem is that a production/live database tuner faces corruption of learning model when it trains over such samples (or collected metrics) which sometimes does not requires any tuning (and this is very much seen on production workloads). However, an offline tuner (which is expected to tune a staging, development, testing landscape database) does not face this issue. This is because in these database the all the queries are batched to form a workload and the workload gets executed for a time frame which generates a high quality sample.

Metrics for Tuner Evaluation. The tuner service the knobs, metrics and provides recommendations for performance improvements. As an end user, the expectations will be to get recommendations when the workload pattern changes in real time and the recommendation should actually improve performance. Currently there are ways in literature which can suggest changes in workload patterns [8], [19]. This works use templates (from queries) and cluster them. However, still there is no such information that with change in workload does the database actually needs a tuning. Secondly, just an increase in throughput cannot be a qualifier to suggest performance improvements. The reason for this is that in production systems, the workload pattern always changes i.e. say the throughput was measured when a query set was executed on a production database. Now after applying the recommendations, the next query set will get changed (or the workload gets changed) and the new throughput cannot be compared with the previous one. Hence, measuring performance based on recommendations on production system is also challenging.

A unifying theme to the above challenges is to identify the actual performance throttling on a database system. With respect to changing workload pattern of users SQL workload, performance throttling detection will help in predicting the incorrect knobs.

The paper makes the following novel contributions:

- **Identifying Performance throttling**: The performance throttling is responsible for identifying the database insufficiency to process SQL queries due to incorrectness of configured knobs. It classifies the knobs into different classes and then, for each class of knobs, it predicts throttling. This module increases the scalability of the tuner deployment by reducing the number of recommendation requests (when compared with the periodic nature of making tuning requests). The underlying services request for recommendation only when a performance throttling is detected. Thus, this module acts as a DBA and identifies when a database requires an actual tuning in real-time. This module identifies performance throttles from relational databases only.
- **Applying Recommendations in an effective way**: The tuning agent running on the database VM/Container as an plugin process, applies the recommendation on service-instance by re-loading the configs. The same plugin is also responsible for tuning of knobs that require a restart of the database.
- **Evaluation**: In production systems as the throughput varies, we need to identify new performance metrics to compare the effectiveness of obtained recommendations. To achieve this, we introduce the number of throttles triggered by the performance throttling module, as a metric.

2 SYSTEM DESIGN

This work presents a generic architecture, which can be easily integrated with any available cloud platform provisioners. As shown in Figure 1, the overall deployment of database tuning service, in an abstract form, is divided into two parts: (i) tuner instances - responsible for executing the ML pipeline to generate new config recommendations and (ii) config director instances - responsible for managing all available customer service-instances. The tuner instances can be spawned via either containers or VMs. There can be more than one tuner instances (also depends upon tuner scalability), where each tuner stores a workload W in a database, where a workload is combination of knob config parameters and metrics observed against those parameters (also called as training samples). Technically as described in [4], a tuner workload W is a set S of N matrices $S : \{X_0, X_1, X_2, ..., X_{N-1}\}$, where $X_{m,i,j}$ is the value of a metric m observed when executing a user SQL workload on database having configuration, j and the workload identifier, i. The tuner service uses the workload W for initial training of both BO and RL style tuners. These workloads are stored in database which is present on a different instance. This database acts as a common central data repository for all tuner instances. Tuning agent runs on the same database (and communicates to DB using Domain Sockets) which is responsible for identifying new workloads and uploads new workloads data periodically to the central data repository. The tuning services running on different IaaS'es, fetch the new workloads from the central data repository. This helps all tuning services to get the new unknown workloads, which might have been observed on a different IaaS, and create a better ML model.

The metric readings and recommendation request calls (we call it something like a tuning request) are event-based and triggered from the performance Throttling Detection Engine (TDE). The TDE gets periodically executed on the database master VM (like a plugin) and triggers recommendation requests to the config director. The TDE runs periodically on the master VM of the underlying database service and is responsible for figuring out performance throttling due to incorrect knob values with respect to current executing user workload. The config director receives the metric data (or queries in case of a RL based tuner) from service instances and triggers recommendation requests to tuner instances. The config director performs load balancing of recommendation request tasks across multiple tuner instances. The Service Orchestrator agent running on database services, is responsible for performing all life-cycle operations of service instances and maintains credentials. When the config director receives a new recommendation for a database service instance from a tuner, the config director passes the new configs synchronously to Data Federation agent (DFA) and Service-Orchestrator, while simultaneously storing it into the config data repository. The DFA

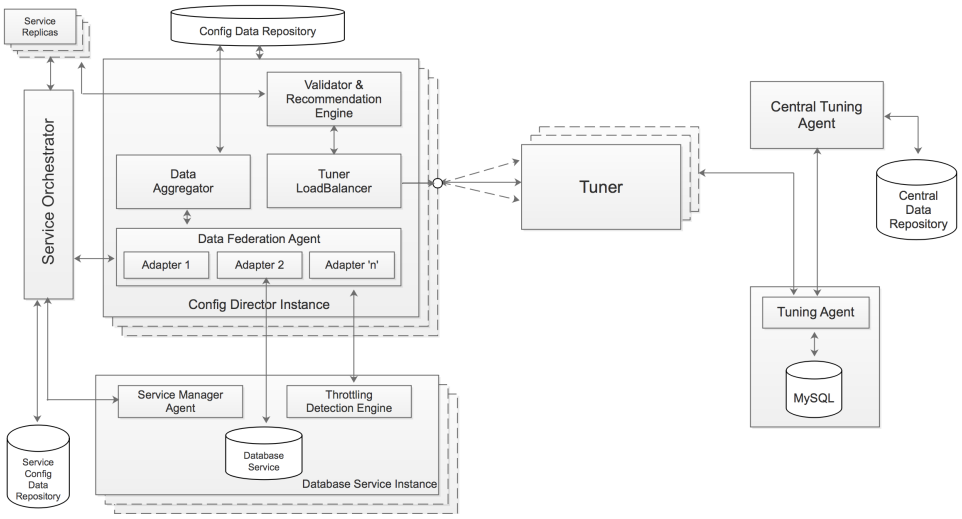


Figure 1: AutoDBaaS architecture.

fetches the credentials from Service Orchestrator layer and hits the APIs of TDE to apply configs to all nodes of the database service (such as all VMs/Containers of the service-instance). The DFA has multiple adapter implementations to get connected to various kinds of database services. As per the architecture, the tuner deployment is capable to tune multiple databases instances (one to many).

2.1 Tuner Instances

The tuner instances as shown in Fig. 1, can be any type BO or RL style tuners. Or can even be a hybrid combination. RL vs BO style tuners is offcourse a debatable topic as both have their pros and cons. BO style tuners when fully tuned with high volume of high quality samples can tune an underlying database in just two to three recommendations, where as RL style tuners need to experience large number of recommendations over the database (as per try-and-error strategy) to learn good configurations for the new workload pattern. At the same time, RL style tuners are highly sclable as they can quickly generate new configurations when properly trained (RL style tuners do not need high volume of high quality samples). Both BO and RL style tuners learning model relay on captured metrics when database actually needed tuning. And metrics/samples captured from database when database did not needed any throughput tuning (this is the often production system case), this corrupts the learning models of tuners. With such wrongly captured metrics BO style tuners face a cascading style corruption and RL style tuners face corrupting the current learning model. This problem of both RL and BO style tuner does not enable them to tune live production workloads, which are often characterized by low throughput or when the database does not needs any tuning. Well this is the major motivation for driving this work.

3 IDENTIFYING PERFORMANCE THROTTLES IN DATABASE

One of the crucial initial steps that the DBA performs before tuning is, monitoring the database to identify whether the database actually needs tuning or not. This module executes periodically as a part of TDE, gathers statistics based on the metrics/features collected using a rule-based approach and identifies potential points when a database needs a recommendation for tuning its configs. As per the proposed performance throttling approach, the config knobs of a relational database can be categorised, based on their properties, into three classes:

- Memory knobs
- Background writer knobs
- Async/Planner estimate knobs

The working of each sub-module is different and is explained as follows:

3.1 Memory Knobs

Memory knobs are the set of knobs which are dependent upon the resource (VM or container) hardware limits. The major portion of memory used by the database is utilised to keep the data in buffer. One of the approaches for identifying throttles could be to find out the actual working database size. To identify this, we use the algorithms proposed by authors in [5] where the authors use gauging techniques to identify the actual working page set. However, the major challenge with this knob is encountered while attempting to update this knob, since it requires a restart of the database. The TDE, collects this information and keeps on sending it to config director instances, where config-director collects the number of throttles and checks the size of the working page set and adjusts this knob value only during the scheduled maintenance downtime.

The other knobs belonging to this segment are related to working area of the database. The knobs related to the working area of memory depends upon the total number of active connections and if it is found to be in-sufficient, then database uses disk or system swap space to perform work operations like sorting, or maintenance operations like index-creation, storing temporary tables, table alter, etc. To get the memory usage details probes needs to be created in the codebase, which is arduous and dependent on freedom given from vendors. Alternatively, figuring out the disk usage while query execution, the query plans can be used as a potential source of information. We use query templating as described in [6], to reduce the total queries (to be examined in production systems), where the queries are converted to a template having a template-id. The queries collected from streaming logs are pre-processed and then converted

to generic templates (having no actual parameters/arguments). The final template selection takes place from the pool of queries by reservoir sampling (for capturing samples from streaming logs) [7]. The selected query templates undergoes execution plan evaluation by substituting the actual (most frequent) parameters to the template. From the plans/streaming logs, it can be easily inferred how much memory/disk the query is going to take. If any of the selected templates (from reservoir sampling) uses disk while execution, signifies that the memory is in-sufficient for execution of queries and now the TDE triggers a memory based throttle signal and asks for knob recommendation from a tuning service (raises a tuning request to the tuner).

However, there can be potential cases when the memory allocated for the buffer is maximum (which means the memory left for other processes becomes less), it is observed that TDE un-necessarily triggers throttle signals. This is a case, where the underlying instance configuration limit is in-sufficient (or the usage has reached the caps limit). When the size of the database is sufficiently higher than the actual memory allocated to database process, it is observed that the TDE frequently triggers throttle signals and is unable to understand that the throttles are being caused because of limited hardware resources. To deal with this cases, we need filtration approach, which identifies such situation and stops the un-necessary throttles (one potential case is the underlying VM hardware resource is in-sufficient and customer needs to upgrade to another plan or ask for more resources for the VM). We face the following challenges when designing such filters:

- There are a specific set of queries which trigger consecutive throttles from one memory knob (like use of aggregate queries triggers a throttle from working memory in PostgreSQL). Situations like this can cause increasing working memory continuously with each recommendation obtained and hence decreasing other knobs (to make room for increase of working memory). However, even after increasing the knob values to the maximum, throttles can get triggered. This situation can easily be captured by rule-based engine and throttles can be filtered.
- For a certain query, consecutive throttles are observed intermittently against different knobs. For example the first two throttles came from working memory and next two throttles came from *maintenance_work_mem*. This becomes very difficult to manage and identify with a rule based approach especially when number of knobs are high.
- There are a specific set of queries which triggers consecutive throttles from more than one set of knobs at a time (like use of aggregate queries, index creation queries, temp table creation queries, etc causes trigger of throttle from multiple knobs). Situations like this are difficult to be captured by rule based engine (becomes more complex when knob numbers increases or is already high) and needs a different approach.

We observed and collected such queries and table shown in Fig. 2 shows the same. In PostgreSQL, working memory is used by the execution engine to perform internal-sorting, joins, hash-tables, etc. We evaluated amount of working memory used by TPCC and CH-Bench, YCSB and Wikipedia bench in absence of indexes. We observed that Wikipedia and YCSB queries do not use working memory (due to absence of complex queries like aggregate, joins, and order-by). The table illustrates the actual working memory allocated and the amount of disk and memory used by queries.

Benchmark – Query Type	Work Mem (in MB)	Disk space utilized (in MB)	Memory utilized (in MB)	Execution Latency (in ms)
tpcc – Scan Query	0.5	3	0.5	8310
tpcc – Scan Query	1	0	0.64	3953
ch-bench – Scan Query	125	260	125	15413
ch-bench – Scan Query	150	0	137	6801
tpcc – Complex Aggregation Query	350	475	350	53013
tpcc – Complex Aggregation Query	375	0	367	19835

Figure 2: Queries and Memory statistics observed on PostgreSQL running on AWS VM, type-t3.x_large

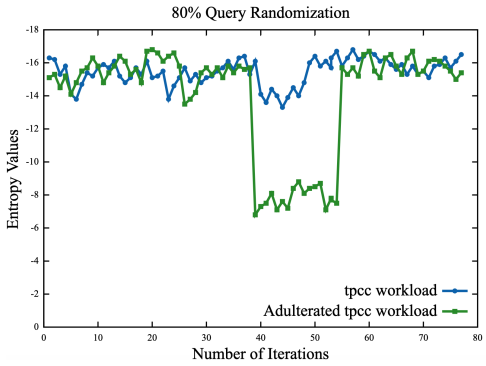


Figure 3: Entropy variation with 80% adulteration probability on Production SQL Workload

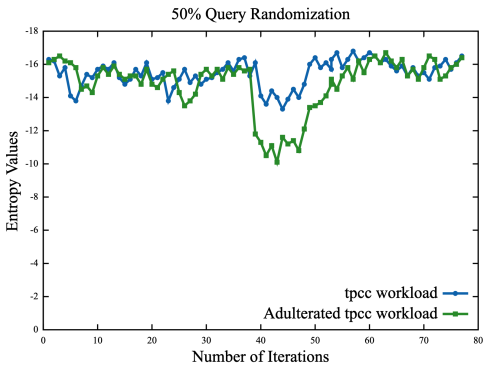


Figure 4: Entropy variation with 50% adulteration probability on Production SQL Workload

In this case, a probabilistic approach is needed to predict the pattern of SQL queries which can cause a potential throttle in performance. The queries which cause more use of working memory are mostly Join, aggregate queries, sorting queries (ORDER BY). On production systems, the frequency of rest queries like index creation or alter table is comparatively lesser. The worst-case scenarios could be all queries are fired with similar proportion. To deal with such cases, or to identify such randomness/query proportion, (to measure the probability distribution) entropy is

used. Entropy of a discreet variable $\mathcal{X}$ with possible outcomes $x_1, x_2, x_3, ..., x_n$ can be defined as:

$$H_n(\mathcal{X}) = -\sum_{i=1}^{n} p(x_i) log(p(x_i)) \quad (1)$$

where $p(x_i)$ is the probability of the i^{th} outcome of $\mathcal{X}$. With a more abstract approach, a generalized entropy can be defined as:

$$\eta(\mathcal{X}) = -\sum_{i=1}^{n} \frac{p(x_i) log(p(x_i))}{log(n)} \quad (2)$$

Value of $\eta(\mathcal{X})$ can range from 0 to 1 i.e. $\eta(\mathcal{X}) \in [0, 1]$. This helps in determining the threshold value of entropy, as for any number of classes the normalized entropy ranges between 0 and 1.

The queries are grouped into specific categories (grouping of obtained query templates), such as Join queries, Select queries, Alter-table queries, Update queries, etc and a hash table is built for each category. The classification of queries is done based on the trigger of throttle from knobs, for example - complex aggregation queries are grouped to one class which triggers throttles to working memory knob. Similarly, we create individual class for each given knob. From the generated logs, we create a hash table containing the class of queries and its frequency. Once the entropy value is evaluated, it can be inferred from multiple observations, that the entropy value is less when high randomness is present or all queries are fired with similar proportion (the query frequency from classes are evenly distributed). This, thus, indicates the SQL queries will, probably, again trigger a throttle (when underlying instance configuration is in-sufficient). However, if the entropy value is high, the degree of randomness is quite less or probability is quite evenly distributed. Thus, provided, if the query class, which is constrained by throttles, has less frequency, it can be concluded that in future, the throttles will not be triggered (as here the limits have not reached the caps and the underlying database depends on the tuner recommendation for knob optimization).

As part of the proposed flow, if more than 8 throttles are triggered consecutively, the entropy value is evaluated, and if the entropy value is higher along-with the memory-knobs reaching maximum cap value, the TDE triggers a plan update (increasing the hardware limits of instance) request to customer and recommendation requests are not sent to config director. Else, it is estimated that the throttles will soon reduce and the same job waits for next 8 throttles before calculating the next entropy value. The graphs shown in Fig. 3 and 4 shows the calculated entropy values while executing TPCC and an adulterated TPCC workload. The TPCC workload was adulterated with index creation, index drop, complex-joins, temp-table creation, order-by and aggregate queries.

In order to showcase the entropy variation, we loaded TPCC with a scale-factor of 18 (which loads around 21GB of data) to Postgresql. However the queries fired mostly hit the working memory and wal-memory knobs
(*sort_buffer_size* in MySQL). The amount of working memory used by TPCC as shown in Fig. 2 is around 0.5 MB, which is quite less to generate a throttle from memory based knobs. Hence now we add complex aggregation queries to TPCC (like queries having heavy sorts), which requires nearby 350 MB. Still we are able to trigger throttle for only working memory using TPCC. Now in order to design such a workload which triggers throttles from all defined classes/knobs, we started adding more queries and procedures, the following queries (analysing from production level performance bottlenecks faced earlier) were added to TPCC bucket:

- complex sorts/aggregation queries - To trigger throttle from Postgresql - *work_mem*, MySQL - *sort_buffer_size* and *join_buffer_size*
- create/delete indexes - To trigger throttle from Postgresql - *maintenance_work_mem*, MySQL - *key_buffer_size* and *sort_buffer_size*
- delete queries: To trigger throttle from Postgresql - *maintenance_work_mem*
- creating temporary tables and firing complex aggregation queries on it to trigger throttle from Postgresql - *temp_buffers* and MySQL - *temp_table_size*

Now the new queries are always added to the actual TPCC bucket based on probability as given in Fig. 3 and Fig. 4 (80% and 50%). With the adulterated TPCC workload, we were able to simulate throttles from all set of classes/knobs. The probably distribution with TPCC varies hugely with the probability distributions of adulterated TPCC due to absence of the new queries and results in entropy difference.

3.2 Background writer knobs

The background writer knobs control the writing of dirty pages from buffer, back to the disk. This write process is triggered by background writer processes or periodic checkpointing processes. However, if the checkpointing process is triggered too often and the amount of data written is high, then it leads to higher values in consumption of I/O throughput and disk latency resulting a decrease in throughput of the database. The other processes involved in writing dirty pages back to disk (background writer) helps in mitigating the same problem, with the aim to reduce the amount of data written by a checkpointing process. Usually the background process writes a fixed number of pages back to disk and the left pages are taken care by checkpointing process. In case of write heavy workloads, the background process writes fixed amount of data causing uncertain amount of data written by a checkpointing process. Given a discrete configuration for the set of knobs, for identifying throttles the following set of challenges needs to be overcome:

- To find out optimal value of checkpointing triggered per unit time. This parameter helps in understanding the overall period till which there can be a surge in disk latency, IO, etc.
- To find out optimal value of data written to disk with trigger of a checkpoint. This parameter helps in understanding the max surge the disk IO and latency parameters can go for write operations.
- There are various processes which write back to disk, for example - WAL writer, statistics writer, log writer, archiver, garbage collector, vacuum. This makes it difficult to figure out holistically the exact amount of data written by checkpointing process.

In order to figure out the exact amount of data written by a specific process requires use of user-level statically defined tracing probes (USDT probes). Where, any low-overhead tracing tool like ebpf or dtrace (Linux Foundation - IO Visor project) can use the probes to get information. The other option is to use kernel-probes (uprobes) for tracing, but this is also independent of the database process levels. Hence the safest way to get this data is to move writing of majority of processes to another

disk. In our experimentation, we changed the disk for storing of WAL, statistics, logs, etc. Now only background writer processes or checkpointing processes and vacuum/garbage-collector processes are responsible for writing on the current disk (where the production database files are located). This strategy also guarantees SLA for minimum IOPS for a disk which stores the actual database and at the same time increases cost of extra hardware and operations. Still the checkpointing process can be interpreted with the vacuum/garbage collector processes which is responsible for updating indexes for dead tuples and defragmenting pages on disk. The frequency of this process can easily be controlled and the left slots can be utilised for monitoring of checkpointing processes. During experimentations, we increased the frequency of vacuum/garbage collector to substantially a higher value and neglect the monitoring of checkpointing during the interval when vacuum/garbage collectors are triggered.

To predict/evaluate the values of optimal checkpointing and optimal amount of data written per checkpoint, the proposed approach uses the historic data of the workloads stored in the tuner's database or in short leverages the tuners experiences of tuning write oriented workloads. The workloads which are generated for tuner, are often pre-generated offline or it considers newer workloads as well (workloads from live database systems). The tuner service for recommending new knob values selects a target workload, and then uses the target workloads data to train the GPR. However, in all cases we monitor the disk latency from external monitoring agents such as Dynatrace. The throttling point for these knobs depends upon the disk latency as the performance degrades when the disk latency increases. In order to figure the optimal checkpoint per unit time with amount of data written to disk, the time difference between peaks in disk-latency is observed and averaged out for consecutive peaks. We define checkpointing per unit time based on the same observations. The checkpointing per unit time is calculated only for the highest observed throughout point in mapped workload.

Each database service in order to get the optimal parameters uses the best information seen/tried by tuner in past. Now, when a throttle is triggered, the tuner maps the current workload $'A'$ (workload representing a target underlying database service) to a target workload $'B'$ which had shown similar features in the past, with respect to the current workload. Now, for B, the timestamp value for the most optimal points observed (with respect to maximum throughput) are captured and passed on to the Dynatrace agent and the disk latency readings are collected. The points are the best recommended knob sets obtained using a trained GPR. From this data point, for the entire duration of workload execution on database, the checkpointing per unit time and respective disk latency is observed. Now on live/production systems, the checkpointing per unit time for A is calculated based on the baseline of disk-latency defined (obtained from tuner) earlier. If in $'A'$ the ratio of checkpointing per unit time and disk latency is more than the ratio of checkpointing per unit time and disk latency for B, then the throttle-detection scripts trigger a throttle signal. However, there could still be scenarios, when the workload A has very less data points (config values vs metrics) and for such a scenario, the mappings are initially incorrect for target workloads. Then for that scenario, the number of throttles could be either more or less, however, with each throttle signal that is triggered, the workload size increases and probability of getting mapped to an optimal workload increases. Thus, the proposed approach eventually improves in efficiency with passing time.

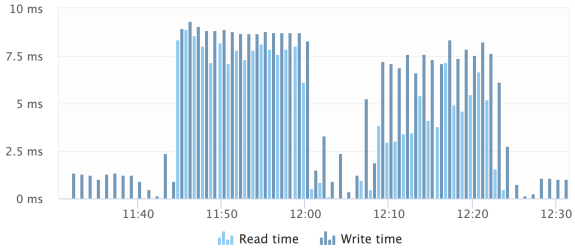


Figure 5: Disk Latency graph for TPCC execution.

The graph, as shown in Figure 5, represents the disk latency incurred when TPCC is executed on PostgreSQL with default knob config values and compared with when it is executed with optimal knob config values. The readings observed from 11:45 to 12:05, show the disk latency values for TPCC execution with default knob values and readings observed from 12:10 to 12:25, show the disk latency values for TPCC execution with optimal config values on PostgreSQL. Here, the TPCC execution on tuned PostgreSQL gives an average disk-write latency of around 6.5 ms and based on this the checkpointing per unit time is obtained. So, this becomes the base line for any workload on live systems which is mapped to the TPCC workload. Here, the major constraint is also that the underlying hardware (storage type as SSD or HDD) should be the same for all systems (databases used for training tuner and live systems).

3.3 Async/Planner estimate knobs

The async knobs are based on the ability of the database to parallelise the query execution, whereas the planner estimate knobs helps the query execution planner to estimate the best route. Most of the database recommends to statically set the planner estimate knobs (random page cost, effective cache size. etc) based on the underlying hardware capabilities. Still it is often seen that increasing/decreasing the values of such knobs (from the recommended values) improves the overall query execution. The async knobs are often defined by the number of parallel worker processes supported per relation by the database. During query execution, the parallel workers are taken from a pool of all defined workers. Often, it happens that the requested workers are not available or it could also happen that setting a higher value for these knobs affects the planner estimates. Thus, it always depends upon the nature of query and to what degree it can support parallel executions.

As this categories of knobs directly or indirectly impacts the planner estimates, it is often required to check the planners cost/benefit optimizations. The straight forward way to trigger a throttle would be to manually increase/decrease the knob values and check the overall cost/benefit optimizations. However, to automate this, the TDE needs to carefully take decision on whether to increase or decrease the value and by how much the value should be increased or decreased. Assuming at a given instance of time, for a given production workload, there exists an optimal values of this knobs. And the optimality does not depends on the underlying hardware (as per recommendations), making this a stochastic environment use case. Reinforcement learning is often seen as the best way for analysing the cost/benefit optimizations of query execution planner [8] [9] [10]. Hence, we model this problem as sequential decision problem and address it by using reinforcement learning.

Hence, we use a very basic Markov Decision Process (MDP) as a very basic RL model to solve the above sequential decision problem. In order tp minimise the uncertainty, the MDP starts with random set of actions and with course of time the action probabilities are adjusted, based on the response from the environment. The RL algorithm tries to optimize an agents returns when the episodes are restricted/limited. The RL engine captures all the queries in a time frame (typically a day or two based on the length of the workload). One episode comprises of atleast 350 to 400 steps (set of actions), where the knob values are changed as per policies (policies are random model initialization) and planner cost/benefit estimates are captured for all queries. The cost benefit estimates are then converted to rewards or penalties.

The TDE uses a MDP to trigger a throttle from this category of knobs. A MDP is represented by $\{Q, A, B, N, H\}$. For all given knob in this category, a MDP is given as follows:

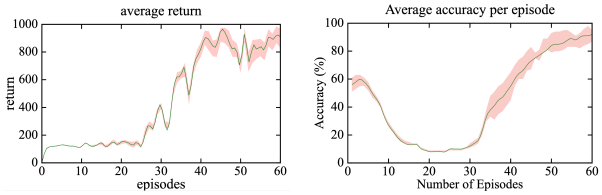
- Q is the finite set of internal states given by $Q = \{q_1, q_2, q_3, ..., q_n\}$, where q_n represents a specific knob value tried before or in current usage.
- $A = \{\alpha_1, \alpha 2, \alpha_3, ..., \alpha_n\}$ is the set of actions performed by the automata (increase/decrease the knob value) where each action has its own probability distribution.
- $B = \{\beta_1, \beta_2, \beta_3, ..., \beta_n\}$ is the response from the environment (cost/benefit calculated from query planner)
- N is a mapping function responsible to map current state and input to the next state and
- H is a mapping function responsible to current state and response to figure out the action to be performed.

The TDE triggers the MDP at interval of 2 to 4 minutes, where the MDP performs cost/benefit analysis by fetching all the queries from log, performing reservoir sampling as described above (in throttling detection for memory knobs). For a given knob value (represented by q_n), based on the action probability, the MDP increases/decreases the knob value by unit step (defined statically). Later the TDE calculates the loss/profit in execution time against the sampled queries with respect to the new knob value and old knob value. If there is a loss, which signifies the action is misleading and the MDP penalises the respective action (which adjusts the probability of the given action α_n) and vice-versa. However if a profit is seen with the change of the knob, the TDE triggers a throttle to get a recommendation from the tuner. The graphs in Fig. 6 presents the learning progress for a production workload as shown in Fig. 8. In the initial episodes, it is observed that the learning is less as the agent is suppose to do more and more exploration of knob configs. However, as the iterations continues, we observe more and more learning (as the episodic rewards increases). This draws a balance between exploration and exploitation.

One can argue that if with the course of time the MDP learns about the optimal/sub-optimal values of the knobs, is it really necessary to go and again ask the tuner to get recommendation. Yes, the tuner needs to be asked as the optimality changes with respect to change in workload pattern and secondly the tuners learning models predict best values for the given knobs by utilising the past seen experiences from set of other production systems.

4 APPLYING RECOMMENDATIONS

The potential challenges in this job could be designing the overall orchestration mechanism for applying these configs, considering the prevalent architecture of the database system like multi-node,



(a) Learning progress of proposed policy (b) Average accuracy of learning process

Figure 6: Measuring Reinforcement Learning accuracy on production workload

high-availability constraints, etc. The configs need to be persisted too such that a database reset or re-deployment doesn't overwrite the settings. Additionally, concerns like how to apply the recommendations without causing a downtime of the running database system, must also be addressed.

Generic Approach. An orchestration approach had to be formulated in-order to apply the recommendations, taking into consideration the above-mentioned challenges. As per the architecture explained in Figure 1, the service-orchestrator is responsible for spawning of database system instances for a customer. The config of the spawned database system is generated and applied initially to the database, by the service-orchestrator. If for any reason (like updating the system, applying security patch, etc.), the database system needs a re-deployment, then the service-orchestrator must re-deploy the system with the updated config of the database.

As per the architecture, the Data Federation Agent (DFA) hits API endpoints of TDE to apply the config recommendations. In case of multiple nodes maintaining high availability, the recommendations are first applied to the Slave node(s). If the process crashes in the Slave node, the config recommendations are rejected. Thus, it is ensured that the Master node is up and the process is still able to serve requests. After the config recommendations are applied to the Master node, the recommendations are stored in the persistence storage used by the service-orchestrator. Thus, whenever the service-orchestrator re-deploys the database system in the future, it retrieves the updated config from the persistence storage. An additional concern here could be the failure in one of the intermediate steps. Since, all of the operations are not atomic, but eventually are expected to yield consistent data (i.e., configs must be same for all master/slave nodes and persistence storage used by service-orchestrator), a reconciler process is defined. The reconciler keeps a watch on config of the database system running on the Master node. If the difference in config is observed for a threshold time-period (watcher timeout), the reconciliation occurs and the config stored in the persistence storage is applied to all nodes. Thus, this eventually leads to rejection of the config recommendation due to error in the intermediate process.

For changing the knobs values, one of the efficient methods is to use Socket Activation (using sockets via *systemd*). This also makes possible to restart the DB since the socket is up and keeps on accepting the incoming requests. However this method only caches the requests but causes a lot of jitter and performance degradation. Another method is to use linux reload signals, upon evaluating this method, we observe very minimal jitter in the performance of the database. A comparative analysis has been

presented in graphs in Figure 7 where the performance of the database is observed under identical load conditions.

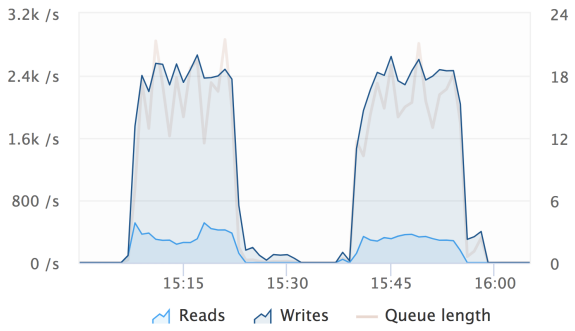


Figure 7: IOPS graph for TPCC execution.

The graphs shown in Figure 7, the TPCC workload is executed with tuned MySQL. Readings during 15:08 to 15:23, shows the TPCC execution without triggering any config reload signals, where as readings between 15:38 to 15:54, shows the TPCC execution which was accompanied by a config signal reload with a frequency of 20 seconds (even with this high frequency of reloads, the performance is not compromised).

Applying Non-tunable Knobs. 'Non-tunable knobs' has been used as a term to categorise such knobs that cannot be applied/tuned without causing a restart of the database process. Since, the restart of database can only be performed during scheduled downtime window (a pre-announced time-period where a re-deployment of database occurs), an approach had to defined for the tuning of such knobs. The design of the approach can be considered for memory-related knobs (as non-tunable knobs are majorly memory knobs). For a non-tunable memory related knob, like buffer-pool's size, the optimum value of this parameter can be obtained from the working set [5]. Once this optimum value is determined, the database system is initially set up with the same value. However, there could be other memory-related knobs that are dependent on such a non-tunable knob. The value of all such knobs must be within the total memory allocated to the database process. Let us consider the following equation. $A + B + C + D < X$ Here, A could be assumed to be a non-tunable knob like $buffer\_cache$ size. B, C, D could be other tunable memory knobs like $work\_mem$, $maintenance\_work\_mem$ and $temp\_buffers$, where X is the total memory allocated to DB process.

There is always an upper limit on buffer-pool knob out of the total memory pool. This knob is changed only during the scheduled downtimes. During the downtime, if the total working page set size is greater than the maximum limit, then we find out the 99^{th} percentile of this knob obtained during all last recommendations before the last scheduled downtime. If the new averaged value is lesser than the current value accompanied by at-least one entropy hit, then this knob value is reduced. The entropy hit indicates that the other tunable knobs have already raised many throttles and now it is mandatory to create more room for tunable knobs by reducing the buffer knob value.

Now when the memory for buffer value is reduced with respect to the current knob value, it increases more room for other memory related tunable knobs. So if the cost on throughput for tunable knobs is more, tuning services rotates around nearly same values for buffer knob, else in the next iteration it increases the value of buffer knob (average value of buffer knob obtained till last last scheduled downtime encountered.)

5 PERFORMANCE EVALUATION

The experiments were conducted on AWS instances, where cloud resources is provisioned by cloud-foundry managed by Bosh. The tuner deployment consists of 12 tuner instances with Ottertune and CDBTune (we do not go for QTune due to unavailability of its codebased in opensource) - m4.xlarge with 4vCPU and 16GB memory, 5 config-director instances - m4.xlarge. We connected a total of 80 live-database deployments (spawned through t2.small, t2.medium, m4.large, t2.large and m4.xlarge VM types) to the tuning. For evaluating the experiments, we used PostgreSQL (v9.6) and MySQL (v5.6). All the tuner instances collected data from one common data-repository (m4.xlarge VM plan) which is shared by all tuner instances. The bare-service-replicas were created: one for each plan and were used to test the recommendations obtained. A real-time customer workload (activity for 33 days) is captured for the purpose of some of the below experiments. The SQL workload has 132 tables, 42.13M queries per day (average), 71K Select queries, 41M Insert queries, 34K Update queries and 0.8K Delete queries with a DB size of 59GB. The query arrival rate is shown in Fig. 8

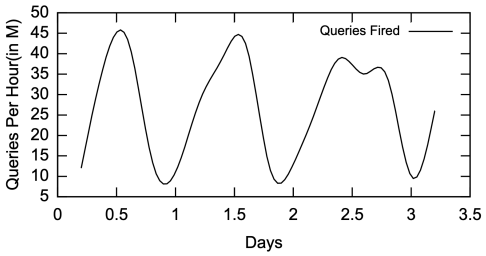


Figure 8: Production workload query arrival rate

Before evaluating the AutoDBaaS on live systems, we perform training of the tuners as per their standard ways [4] [18]. The first experiment we design is to measure the tuning requests per second on production landscape where both ottertune and CDBTune is being used for tuning. Figure 9 showcases one such outcome to illustrate the impact on scalability challenges of a BO style tuner. When comparing scalability of BO vs RL style tuners, a RL style tuner generates new configs very fast (but sometimes takes a long time to come around a good configuration). As per the BO approach, generating a new configuration takes around 200 seconds (which is assumed to be a good configuration). Both the RL or BO style tuners follow periodic approach (with a periodic length of nearly 5 to 10 minutes). In this case we bring in TDE which breaks down the periodic tuning approach. We measure the requests per second for live databases on production landscape where we compare requests per seconds generated when TDE checks in, periodic approach with a period of 5 min and periodic approach with a period of 10 mins. In both the cases it seems like the TDE approach gives a reduction and comes to peak when the workload pattern changes a lot like say morning 8AM to 11AM (when most of the microservice usages surge). The tuning requests per seconds when TDE checks in also directly gets impacted by the efficiency of tuner being used. If the tuner generates good configuration, in the next upcoming iterations, there are pretty less chances of a throttle getting detected.

As the proposed work largely reduces the tuning requests per minute, this evidence seems to directly impact the scalability of the tuning services and specifically the BO style tuners.

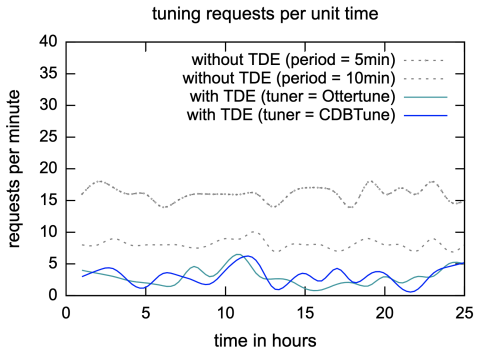


Figure 9: Requests per minute graph for 80 live connected databases

Next we measure the performance throttles due to incorrect knobs for some standard and production workloads. The parameters for the standard workloads was for (1) tpcc, 3300 requests per second with 26 GB of database size, (2) wikipedia, 1000 requests per second with 12 GB of database size (3) twitter, 10000 requests per second with 22 GB of database size and (4) ycsb, 5000 requests per second with a database size of 20 GB. We used oltpbench to do the benchmarking on postgresql (Fig. 10) and mysql (Fig. 11) on m4-large instances. In order to purely measure the performance throttles, we do not go for a tuning session. These throttles presents averaged score for nearly 20 to 25 iterations. However, for production systems (as described above), we do not run iterations, rather they are actually captured from live systems directly for the workload described above and measured at different timestamps. We observe that for both postgresql and mysql, the write heavy workloads raise more throttles for background writter knobs, read-heavy/mix workloads raise more throttles for memory and async/planner knobs and for production workload it seem like a mix of ratios.

The next crucial metric for evaluation is Performance. There could be two metrics for measuring the performance of the proposed approach: (i) the throughput of the database system and (ii) the number of throttles encountered by the database system.

In Fig. 12, we measure the average throughput on live database using (1) Ottertune and (2) Ottertune with TDE (i.e. Ottertune only captures high quality samples from TDE). Ottertune uses samples from both production-workloads and offline-workloads (like executing tpcc offline) to train GPR. As per the tuning pipeline of Ottertune, it bootstraps with offline-workloads and starts tuning live systems. It separately captures each experiences from each workload (i.e. eiter live or offline). There is no chances of training model corruption with offline workloads as samples from offline workloads are captured are always of high quality (i.e. there is no such point when a offline workload does not requires a tuning). So when new batches of production system are hooked with the same ottertune instances, ottertune's throughput is roughly the same as compared with Ottertune + TDE as initially Ottertune uses offline samples (i.e. samples taken from offline workloads) to train GPR. However, the samples captured from the first batch of productions systems causes corruption to GPR (with high probability). Hence, when such samples are

Experimental Setup			
Variable used in Fig. 14	workload	Metrics window length	knobs class
#1	YCSB to TPCC	5 min	background writter, async/planner
#2	TPCC to YCSB	5 min	memory, async/planner
#3	YCSB to Wiki	7 min	async/planner
#4	Wiki to YCSB	5 min	NA
#5	TPCC to twitter	6 min	memory, async/planner
#6	Twitter to TPCC	5 min	background writter

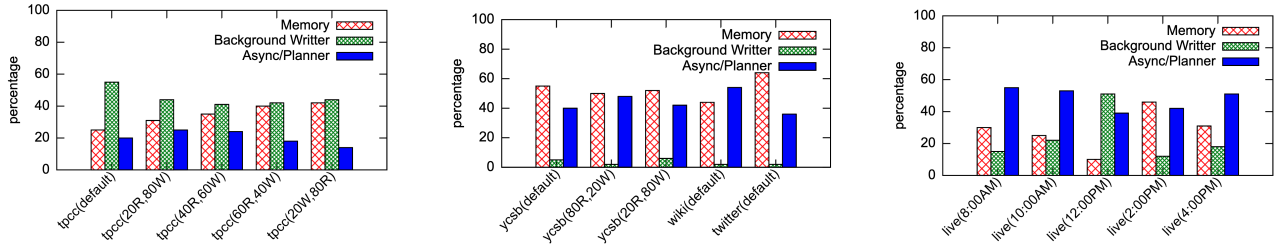
Table 1: Experimental parameters and values

utilized to tune other set of production systems, the accuracy of GPR recommendations is extremely low. Hence, in Fig. 12 we hook in the 40^{th} database instance and measure the throughput. As shown in the graphs the proposed approach seems to perform well and the main reason for that is there is no possible learning corruption in learning. For the workload executing in this database, we observed that Ottertune mapped the workload (with high mapping scores) to nearly 14 different workloads (to leverage tuning experiences) where only 4 of them were offline workloads. Similarly, we measure the same set of throughput when CDBTune is used as tuner. Here as CDBTune does not so much utilizes past learning experiences or atleast the way Ottertune does it. CDBTune minimally utilizes offline training but for sure does not uses learning from other production tuning experiences. Therefore in the case of CDBTune, this problem happens directly from the first hooked/subscribed database. The graph shown in Fig. 13 presents the throughput measured on the first database connected to CDBTune.

We also measure the effectiveness of performance throttling with changing of workload pattern by execution of standard workloads. The graph shown in Fig. 14 presents the same. This experiment is designed to measure how throttling detection helps to quickly capture workload change. In this experiment, we loaded 22GB of TPCC data, 24GB of TPCH data, 18.34 GB of YCSB data, 16 GB of twitter data and 20.2GB of wikipedia data on a m4-xlarge instance of postgresql. And we measure the throttles detected upon change of queries (i.e. from one workload to another). We present the details of experiment done here in the below table 1:

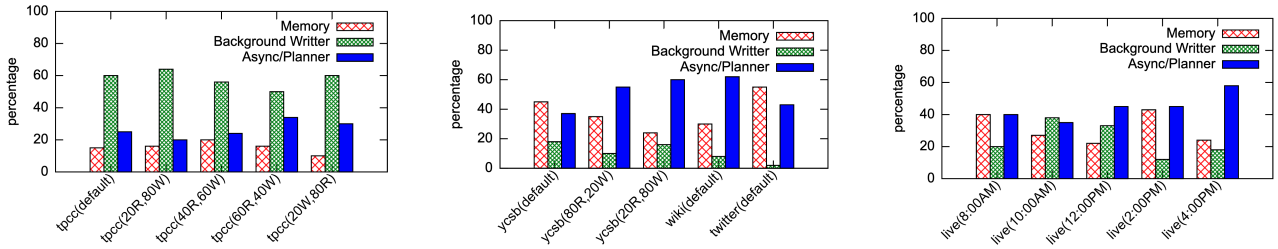
In this experiment we also observe and present the class of throttles. The tuner has a direct impact on the total number of throttles. This is because a single throttle triggers a tuning request and tuner recommends back a good configuration. Hence, in case of a very idealistic tuner the underlying database should not trigger more than one throttle as the idealistic tuner is expected to get the best config which would cause no throttles in the next iteration.

Lastly we tried measuring the accuracy of the throttling detection engine for all the classes of knobs what we presented. To evaluate the accuracy of throttles raised, either we could have used the human knowledge, where an administrator would have verified each throttles manually. But, this approach could have been time-consuming and could might result with a biased decision of the administrator. So another way to evaluate the throttles was to use the tuning of an already trained tuner. We trained Ottertune with offline workloads like TPCC, YCSB, Wikipedia and Twitter and then observed the throttles classes and configurations generated by Ottertune. If Ottertune recommends a majority of knob (say out of top 5 ranked knobs) whose class is



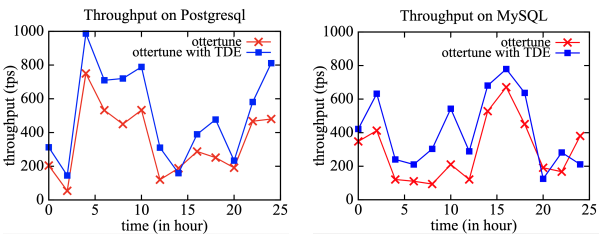
(a) write-heavy workload (b) mix/read-heavy workload (c) production workload

Figure 10: Performance Throttles detected on postgresql for varied set of workloads



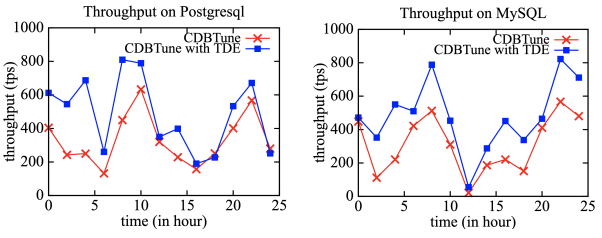
(a) write-heavy workload (b) mix/read-heavy workload (c) production workload

Figure 11: Performance Throttles detected on mysql for varied set of workloads



(a) Average throughput per hour on Postgresql (b) Average throughput per hour on MySQL

Figure 12: Throughput graph for live production database with Ottertune



(a) Average throughput per hour on Postgresql (b) Average throughput per hour on MySQL

Figure 13: Throughput graph for live production database with CDBTune

same as the class of throttle, then the throttle was accurate else we consider the throttle to be not accurate. This is specifically tested on the same workload with which Ottertune was trained i.e. TPCC, YCSB, Wikipedia and Twitter (as for the same trained

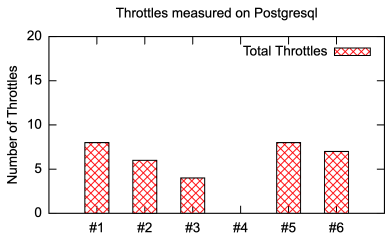


Figure 14: Throttles captured when tuner is Ottertune

data accuracy would be very high). Ottertune recommendations sometimes perform exploration for the Gaussian Models better training. We minimize this exploration by setting appropriate hyper parameters manually. With this settings, Ottertune's recommendation should least explore and only aim to maximize the throughput. We loaded similar amount of data on a Postgresql m4-xlarge instance as done in the previous experiment. The graph shown in Fig. 15 presents the same. We observed high accuracy for memory and background writter knobs and a lower accuracy for planner/async knobs. However, we are confident more for planner/async knobs as the throttle points clearly shows improvement as per cost-benefit analysis of planner estimates. However, we observed ottertune fails to understand such throttles mainly because of absence of planner estimates in the metric set that it captures for postgresql.

6 RELATED WORK

Facebook introduced Pressure Stall Information (PSI) [11] for evaluation and control of computational resources across large data centers. It is one of the first canonical ways of measuring resource pressure increase as it develops based on pressure metrics such as memory, CPU and I/O using *cgroup2* and *oomd*. There

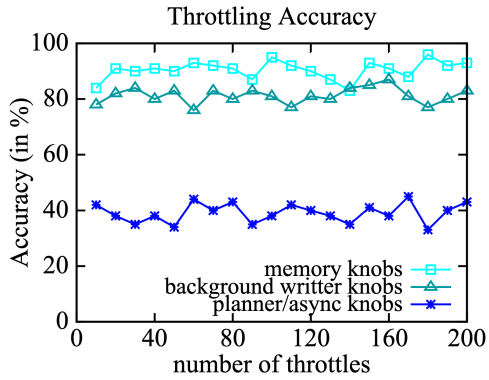


Figure 15: Accuracy of performance throttles on Postgresql

are many methods which exists in literature which try to capture similar pressure on computational resources. However in case of databases these approach seems to be in-efficient in figuring out pressure on individual database knobs. This is also because of the reasons that database knobs are mostly indirectly-non-linearly dependent on the computational resources.

Oracle came up with a database internal monitoring mechanisms [14] and [15] to identify the bottlenecks in performance due to misconfigurations in internal components or knobs. Here, authors propose 'database time' of query as a parameter to figure out performance bottlenecks. Later this information is given to DBA's for tuning of knobs. The system uses heuristics from performance measurements for tuning of memory knobs and however does not tunes all set of knobs.

In literature, there are many knob tuning approaches [12] and [13], which are either specific to specific databases or tune only a subset of knobs. Other PaaS providers like AWS RDS service, gives freedom to consumers to tune it based on the workload. The architecture of Ottertune seems to meet the requirements of PaaS tuning offerings to customers, based on its capabilities to tune multiple databases by leveraging the workloads seen by tuner in the past. Oracle came up with autonomous database, a similarly solution came from Microsoft, however the approaches does not tune more than one database (unable to leverage experience gained by another system) and is only coupled to tune one database at a time (and thus increasing cost of tuning).

Also there exists multiple works that have focused on tuning database knobs. However, there exists two main classes: (1) Search based methods like BestConfig [16] and (2) Learning based methods which includes BO style learning or RL style. We do not consider Search based methods for tuning production systems as it expects the user to execute the workload on staging landscapes and tries tuning it. However, it cannot tune live systems as it takes huge amount of time to get to a good configuration. This is because for every tuning from scratch they again start the searching process from scratch. This work mainly focuses to solve specific problems of learning based methods like Ottertune and CDBTune.

7 CONCLUSION

We presented a generic tuning architecture for tuning services to be provisioned by any PaaS model. In this work, we bring in the challenges and drive them to make the AutoDBaaS more robust for production environments. To take up all the challenges, this work presented (1) methods to monitor database and detect performance throttling, which helps the database to trigger recommendation requests only when potentially required and calculating the monitoring/observation time, (2) methods for applying and validating the obtained recommendations on production systems. Lastly we evaluate the proposed architecture on cloud-foundry managed by Bosh running on AWS. With our approach of detecting performance throttling, we were able to achieve better scalability. On Production systems, due to varying load - throughput, we measure the performance of tuning recommendations in terms of performance throttles hit on production systems. As the existing learning based method needs high quality samples from production system, proposed throttling detection engine enables it to do so. Hence, in this work we also achieve better throughput as throttling detection approach reduces corruption of learning methods.

In the coming future, we would like to explore more on using reinforcement learning methods to capture the performance throttles and making the current TDE free from static rules.

REFERENCES

[1] A. J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra. Adaptive Self-tuning Memory in DB2. In VLDB, 2006.
[2] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer. Using probabilistic reasoning to automate software tuning. In SIGMETRICS, 2004.
[3] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. H. Tung. A new approach to dynamic self-tuning of database buffers. ACM Transactions on Storage, 4(1), 2008.
[4] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In Proc. of the 2017 ACM International Conference on Management of Data. ACM, 1009–1024.
[5] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In SIGMOD Conference, pages 313–324, 2011
[6] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based Workload Forecasting for Self-Driving Database Management Systems," in Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 631-645.
[7] J. S. Vitter. "Random sampling with a reservoir". ACM Transactions on Mathematical Software (TOMS), 11(1):37–57, 1985.
[8] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. "Self-driving database management systems". In CIDR, 2017
[9] A Pavlo, Evan P. C. Jones, and S Zdonik. 2011. "On predictive modeling for optimizing transaction execution in parallel OLTP systems". in Proc. VLDB Endow. 5, 2 (October 2011), pp. 85-96.
[10] D Basu, Q Lin, W Chen, H Tam Vo, Z Yuan, P Senellart, and S Bressan. "Cost-Model Oblivious Database Tuning with Reinforcement Learning". In Proceedings, Part I, of the 26th International Conference on Database and Expert Systems Applications - Volume 9261 (DEXA 2015), New York, NY, USA, pp 253-268.
[11] Facebook PSI, https://facebookmicrosites.github.io/psi/docs/overview
[12] B. Debnath, D. Lilja, and M. Mokbel. SARD: A statistical approach for ranking database tuning parameters. In ICDEW, pages 11–18, 2008.
[13] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with iTuned. VLDB, 2:1246–1257, August 2009.
[14] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis and tuning in oracle. In CIdR, 2005.
[15] S. Kumar. Oracle Database 10g: The self-managing database, Nov. 2003. White Paper
[16] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In SoCC. ACM, 338–350.
[17] Guoliang Li, Xuanhe Zhou, Shifu Li, Bo Gao. QTune: A QueryAware Database Tuning System with Deep Reinforcement Learning. PVLDB, 12(12): 2118 - 2130, 2019.
[18] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, and et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In SIGMOD, pages 415–432, 2019.
[19] B. Mozafari and et al. Performance and resource modeling in highly-concurrent oltp workloads. SIGMOD, pages 301–312, 2013.

Industrial Paper

Scalable Spatio-temporal Indexing and Querying over a Document-oriented NoSQL Store

Nikolaos Koutroumanis
Deptartment of Digital Systems
University of Piraeus
Piraeus, Greece
koutroumanis@unipi.gr

Christos Doulkeridis
Deptartment of Digital Systems
University of Piraeus
Piraeus, Greece
cdoulk@unipi.gr

ABSTRACT

In this paper, we provide an in-depth study of the performance of spatio-temporal queries in document-oriented NoSQL stores. Existing NoSQL stores provide limited support for spatial data and (quite often) no native support for spatio-temporal data. As a result, the performance of query execution over large collections of spatio-temporal data is often suboptimal. We present an approach for indexing spatio-temporal data, which is applicable to *any* NoSQL store that provides key-based access to data *without modifications to its code*, and we show how to generate data partitions that preserve data locality. Moreover, we show the impact of indexing and partitioning on the number of cluster nodes that serve a query, and we discuss the advantages and disadvantages for different applications. We adopt a methodology for the evaluation of spatio-temporal range queries, which can serve as a benchmark. In our experiments, we focus on MongoDB (as a representative NoSQL store that provides spatial support) and we study the impact of indexing spatio-temporal data on performance, using both real-life and synthetic data sets in a medium-sized cluster.

1 INTRODUCTION

Big spatio-temporal data sets are collected every day at unprecedented rates [15, 17], due to emergent applications, such as fleet management solutions, surveillance systems in maritime and aviation, human and animal tracking, IoT sensor feeds, location-based web search, and social networks with geotagged content. These applications generate huge volumes of positional information represented as points, which require scalable storage and retrieval, so that data analysis techniques can be applied to discover hidden spatio-temporal patterns. As a result, scalable spatio-temporal data management is a challenging research topic, and efficient solutions are required for storage, indexing and querying.

NoSQL stores [4, 7] comprise the state-of-the-art in scalable storage to date. However, while support for spatial data is provided recently by an increasing number of NoSQL stores, this is seldom the case for spatio-temporal data. In fact, even spatial data access methods are not always optimized in today's mainstream NoSQL stores. While most relational DBMSs have adopted R-trees [11] (or its variants [2, 16]) for efficient spatial indexing, NoSQL stores with spatial support adopt GeoHashes to map spatial data to one-dimensional (1D) values, which is then indexed using traditional 1D indexes, such as B-trees [6] (see Table 1). Our conjecture is that this decision relates to the cost of building and maintaining a distributed R-tree. Consequently, the performance of existing solutions is suboptimal, when faced with the challenge of efficient and scalable retrieval of spatio-temporal data.

	Database	Spatial Indexing
RDBMS	PostgreSQL (PostGIS extension)	R-Tree
	MySQL	R-tree
	Oracle	R-tree
	MariaDB	R-tree
	SQL Server	B-tree
	SQLite (SpatiaLite extension)	B-tree
NoSQL	MongoDB	B-tree
	Redis (Geo API)	Sorted Set
	DynamoDB	B-tree
	Elasticsearch	BKD-tree
	Neo4J	B+Tree

Table 1: Spatial support in most popular relational and NoSQL data stores

Our work is motivated by real-life applications, revolving around fleet management operators in the urban domain, which collect large volumes of positional data from GPS-equipped vehicles daily. The specific use-cases that are supported by our work relate to exploratory analysis of historical routes, using multiple spatio-temporal queries of varying granularity. The retrieved trajectories are analyzed for fleet cost reduction (by analyzing the fuel consumption of historical routes), intelligent routing, as well as for discovering movement patterns. The challenge is to provide a scalable storage and spatio-temporal querying solution for large volumes of historical mobility data. Unfortunately, existing industrial solutions are not optimized for spatio-temporal querying at scale, thus fleet management operators apply data analysis techniques only on recent subsets of their historical database, while older data is kept in cold storage.

Motivated by these limitations, in this application paper, we provide an in-depth study of querying spatio-temporal data at scale, focusing on a document-oriented NoSQL store, namely MongoDB. The choice of MongoDB is justified due to its wide popularity among big data developers, and its maturity compared to other competitive technologies. We explain the internal details of indexing and sharding, focusing on how spatial data is supported, and eventually design a solution for spatio-temporal data using the built-in indexes of MongoDB. Then, we propose an alternative approach that uses the Hilbert space-filling curve (which has been shown to have nice clustering properties [14]) to generate one-dimensional (1D) keys, which facilitates indexing of spatio-temporal data, and allows to preserve data locality in the nodes of the MongoDB cluster. Moreover, this approach can be implemented *on top of* MongoDB (and other key-based NoSQL stores), thus being directly applicable for any application.

In particular, our approach has an effect on sharding, essentially creating spatio-temporal data partitions that preserve data locality. As we demonstrate in our empirical evaluation, this has a profound impact on performance.

Our contributions can be summarized as follows:

- We propose an approach for efficient storage and querying of spatio-temporal data based on Hilbert encoding, which can be applied to any NoSQL store that supports key-based access to data.
- We show that our approach achieves spatio-temporal data locality across the distributed data partitions, and we discuss the advantages and disadvantages for different applications.
- We present a methodology for evaluating the impact of spatio-temporal access to data at scale, which can also serve as a benchmark for spatio-temporal queries in NoSQL stores.
- We perform extensive experiments over a MongoDB installation on a public cloud, and we study the effect of different metrics (such as keys and documents accessed, nodes involved in query execution) on execution time using both real and synthetic data.

The remainder of this paper is structured as follows: In Section 2, we review related research efforts. Then, in Section 3, we present the internal mechanisms of MongoDB for indexing and handling spatial data. Section 4 outlines our approach for indexing spatio-temporal data. Section 5 presents the results of our empirical evaluation, and Section 6 concludes the paper.

2 RELATED WORK

Spatial data indexing is a long-studied topic, with R-tree [11] and its variants [2, 3, 16] being a prominent data structure in centralized databases. Even though approaches for distributed R-trees have been proposed [8], they have not been adopted by today's NoSQL stores, probably due to the high maintenance cost in distributed settings and due to the gradually diminishing performance after many inserts/updates.

2.1 One-dimensional Indexing of Spatial Data

Space-filling curves have extensively been used in spatial databases in order to map high-dimensional data to one-dimensional values, which can be indexed using standard data structures, such as the B-tree. Although many variants exist, notable examples include the z-order and the Hilbert curve (depicted in Fig. 1). In the context of data management and indexing, the objective of all space-filling curves is to preserve data locality in the one-dimensional space, so that spatial range queries can be transformed to one-dimensional intervals of small length, in order to reduce the number of false positives.

Even though the idea of GeoHashes was first proposed by G.Niemeyer in 2008, it bears similarities with space-filling curves (in particular with z-order), which have been known for decades. MongoDB uses GeoHashes to store spatial data efficiently. The idea of GeoHashing is to use a hierarchical subdivision of the 2D spatial domain, which uses multiple layers, with each layer divided in a set of cells (or buckets). At the top layer, four buckets exist which are derived by splitting each dimension in the middle. Then, each bucket at the top layer can be represented by 2 bits: 00, 01, 10, and 11. The hierarchical subdivision is performed recursively, so at the next layer sixteen buckets exist and four bits are used to address a bucket. As a result, any 2D spatial point

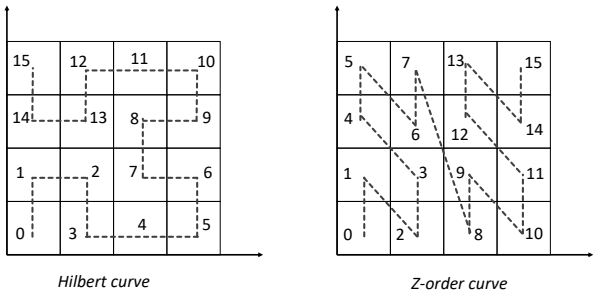


Figure 1: Illustration of the Hilbert and z-order space filling curves

is assigned into a lower-level bucket uniquely, and the bit representation of the bucket can be used as an indication about the location of the point in the 2D space. The more layers, the higher the precision of the respective location. Finally, GeoHashes use a *base32* String representation, instead of a bit representation, which uses a 32-character set comprising the twenty-six letters a–z, and the digits 2–7. As an example, Athens (Greece) has coordinates (37.983810, 23.727539) which is represented as a GeoHash of "swbb5ftzes" for precision of 10 characters. If we used lower precision, the corresponding prefix would be obtained. For instance, the GeoHash of Athens for precision of 5 characters is "swbb5".

2.2 Spatial and Spatio-temporal Queries in MongoDB

NoSQL systems are widely used by modern applications for scalability and high performance. For a recent survey on NoSQL stores, we refer to [7] and also to the early work of Cattell [4].

There exists some work on studying the performance of built-in mechanisms for spatial query support in MongoDB. Duan and Chen [9] compare MongoDB against ArcGIS, in order to assess the performance of the spatial extension of MongoDB against a well-established GIS. More recently, Bartoszewski et al. [1] compare MongoDB against PostGIS for spatial data. However, both studies evaluate the systems on a single machine, which is a limitation because it hides the impact of distributed storage on query execution.

In [13], an experimental evaluation of MongoDB against PostgreSQL is performed for spatio-temporal data. This is one of the few studies that try to evaluate MongoDB's capability in terms of querying spatio-temporal data. However, their study has some limitations, most notably the lack of data partitioning in evaluation. Instead, a small cluster is used and all machines contain replicas of the data set. In contrast, our work provides an in-depth investigation of different aspects of spatio-temporal data management, including indexing, data partitioning and load balancing, in a much larger deployment of MongoDB in a sharded cluster.

ST-Hash [10] follows an approach for spatio-temporal indexing on top of MongoDB. The main idea is to extend GeoHashes in a way that time is also incorporated in a string representation of a one-dimensional value. This value can be decomposed to obtain the corresponding spatial and temporal information. Hence, a one-dimensional index is built on this string value, in order to support efficient point and range searches. However, the resulting encoding uses the year as a prefix, which is not

effective for certain query types. For example, queries with high spatial selectivity but low temporal selectivity cannot exploit the encoding, in order to efficiently identify which data blocks need to be accessed.

SIFT [12] is an implementation of a distributed spatial index upon MongoDB. The study focuses on the ingestion, indexing and querying of highly-skewed spatial data. The basic data structure of SIFT is based on a tree where the spatial objects are represented by their minimal bounding boxes. The tree structure follows an approach so as to avoid any rebalancing, splitting and merging operations when spatial objects are inserted to the index. Nonetheless, the index is limited only to spatial queries.

3 BACKGROUND ON MONGODB

MongoDB [5] is a popular document-oriented NoSQL store that stores data in the form of *documents* in *collections*. *Documents* are binary JSON objects (BSON) that consist of a set of fields with associated values. Values can be simple or complex, e.g., an array or even a nested document. Each document is typically associated with a key that uniquely identifies it. *Collections* are containers for conceptually similar documents, however no restrictions are imposed to the schema of each document. In MongoDB, collections are stored in *databases* which are namespaces for physical grouping of collections.

3.1 Indexing in MongoDB

The main indexing structure used in MongoDB is the B-tree [6], which supports both point and range queries. Apart from *single field* indexes, which index documents based on a single field, a *compound* index can be used to combine multiple fields (up to 32 fields), thus supporting queries with predicates on multiple fields.

Compound indexes are organized hierarchically based on the *declared order* of the *index keys*. In the case of two fields A and B, the compound index first organizes the sorted values of A in buckets, and then these buckets keep references to buckets that hold the sorted values of B. An example is depicted in Fig. 2, where A=`hotelName` and B=`price`, denoted as `{hotelName:1, price:1}` in MongoDB. This indexing scheme has some important consequences on performance. First, only queries with a predicate on A can benefit from this index, since the value of the predicate is necessary to start the traversal of the index and locate buckets with relevant keys efficiently. Second, it is beneficial to use as first index key a field that has many distinct values, in order to effectively narrow down a search to few buckets only. As a result, the order of index keys in a compound index determines the performance of searches.

MongoDB holds by default a field for each document, called `_id`. The field represents the identifier of a document and is unique, acting as a primary key. The default type of its value is *ObjectId* with 12 bytes. Its length consists of 4-byte timestamp value based on ObjectId's generation, a 5-byte random value and a 3-byte incrementing counter which is initialized to a random value. For the `_id` field, MongoDB maintains a single-field index which cannot be dropped.

3.2 Indexing Spatial and Spatio-temporal Data

Two variants of spatial indexes are supported in MongoDB; a *2d* index, which manages data on a two-dimensional plane, and a *2dsphere*, which calculates geometries on the surface of the earth.

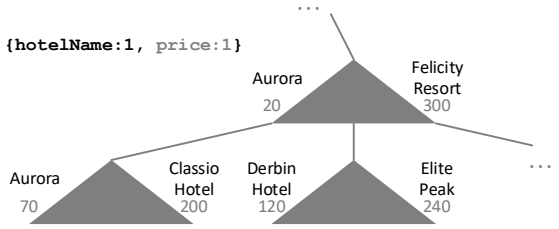


Figure 2: Example of a compound index on fields `hotelName` (string) and `price` (integer). The documents are organized based on `hotelName` and then based on `price`.

Both of them are applied on fields whose values hold spatial data. The values must be either *GeoJSON* objects or *legacy coordinate pairs* which is a representation of the longitude and latitude values either with the usage of two-sized array or by embedding the elements in a document. The spatial indexing mechanism in MongoDB is based on GeoHash[1], where a hierarchical subdivision of the 2D spatial domain using multiple layers takes place. The cells that result from the division of the space are represented by bits. The more bits that represent a space, the higher the precision of the respective location. The GeoHash values that are stored in the index consist of 26 bits by default. They can be set up to 32 bits, performing better for spatial queries, but at the expense of occupying more space in memory. A spatial index can be combined with another field by means of the compound index.

Unfortunately, indexing spatio-temporal data is not directly supported in MongoDB. As a result, our premise would be to build a compound index over the fields storing the spatial information and the temporal information respectively.

3.3 Sharding

Sharding refers to data partitioning and assigning the obtained partitions to MongoDB servers, also called *shards*. Specifically, when sharding a MongoDB collection, its documents are distributed across shards based on a *shard key*. When defining a shard key, MongoDB separates the range of shard key values into smaller non-overlapping ranges with continuous keys. Each of these ranges are associated with a *chunk* and contain a subset of the sharded collection. Also, a chunk has a configurable size which is 64MB by default, and if exceeded, it is split.

The configuration of small-sized chunks leads to a more even distribution of data. However, migrations become more frequent, adding overhead to the network and to the query routing layer (known as *mongos*). Large-sized chunks enforce fewer migrations at the expense of a less even distribution of data. MongoDB achieves load balancing through the (re-)distribution of chunks among shards. The *Balancer* runs in the background so as to migrate chunks across the shards, targeting to achieve an even distribution of chunks in the cluster.

Apart from the definition of the shard key, the sharding operation of a collection requires the strategy type which can be either *range* or *hashed*. With *range sharding*, it is highly probable that documents with similar shard keys will be in the same chunk or shard, as depicted in the example in Fig. 3. This enables

[1] https://docs.mongodb.com/manual/core/geospatial-indexes/

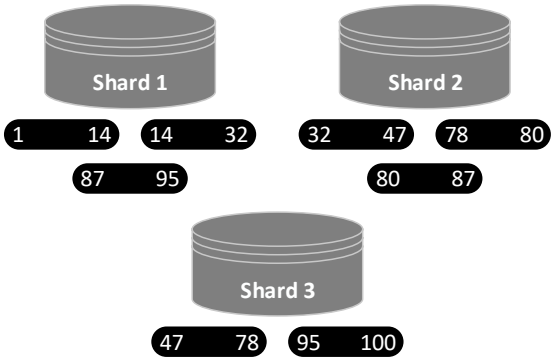


Figure 3: Instance of chunks' distribution that result from a range sharded collection, on a field that contains values from 1 to 99.

routing range queries to specific shards only. On the other hand, in *hashed sharding*, chunks or shards are unlikely to contain documents with similar shard key values. This may serve well for cases where broadcast operations are preferable.

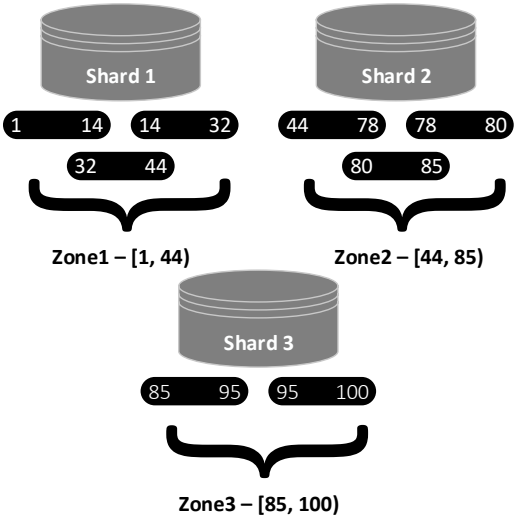


Figure 4: Instance of chunks' distribution that result from the definition and assignments of zones: $[1, 44)$**,** $[44, 85)$**, and** $[85, 100)$ **on the shard key.**

Sharding imposes the creation of a single or a compound index on each shard, based on the field/s of the shard key. MongoDB supports manual grouping of documents based on ranges of shard key values through the concept of *zones*. A zone can be associated with any shard. Similar to chunks, zones have lower inclusive and upper exclusive boundaries and their covering ranges do not overlap. By associating a zone with a shard or shards in an already sharded collection, the cluster migrates the affected data to the respective zones, if a collection is already sharded. Division of chunks may occur so as to follow the data distribution scheme determined from zone(s). If zones are set up before sharding a collection, chunks are created for the defined zone ranges (and additional chunks if necessary) to cover the entire range of the shard key values. Fig. 4 shows an instance of storing the data using zones. In the figure, each shard is assigned with one zone, thus shards maintain more contiguous key ranges.

4 INDEXING AND PARTITIONING SPATIO-TEMPORAL DATA

In this section, we present two alternative approaches for indexing spatio-temporal data, with different implications on partitioning and eventually on the performance of query execution. Our work focuses on point data, which covers many life real-applications, and we leave other data types (such as polylines and polygons) for future work. The first approach relies on the built-in features of the spatial extension of MongoDB for indexing and time-based sharding, therefore we consider it as a baseline solution. Then, we propose an alternative approach that adopts the Hilbert curve to map data to 1D values that will be stored and indexed as a single field, also enabling sharding based on spatio-temporal criteria.

4.1 Baseline Solution

4.1.1 Indexing. The baseline solution for efficient querying of spatio-temporal data stored in MongoDB is via the usage of a *compound* index that is built on the spatial and temporal fields of the documents. Consider the following document structure:

```
{
    "_id": 1,
    "location": {
        "type": Point,
        "coordinates": [37.983810, 23.727539]
    },
    "date": ISODate("2018-10-01T08:34:40.067Z"),
    ...
}
```

The spatial field (`location`) is supported by the built-in *2dSphere* index, and is combined under the *Compound* index with the `date` field. As a result, two possible solutions for spatio-temporal indexing can be designed by exploiting directly the built-in features of MongoDB, which differ on the order of fields: (`{location, date}` and `{date, location}`). The former approach favors the spatial dimension and organizes the buckets of the temporal dimension under the ranges encoded by GeoHash values. The latter approach favors the temporal dimension and organizes the buckets of the encoded GeoHash values under the ranges of the temporal values. As a result, the first approach works well for queries with high selectivity in the spatial dimension, whereas the second approach is more suitable for queries with high selectivity in the temporal dimension.

4.1.2 Sharding. In order to adapt the baseline solution to a distributed environment, a sharding key needs to be defined. We opt for setting as shard key the `date` field. In this way, the spatio-temporal queries are expected to access only specific shards based on the temporal constraint, thus avoiding broadcasting, i.e., routing the query to all shards. In MongoDB, broadcast operations occur if a query's field constraints are not found in the shard key[2]. It should be mentioned that the spatial field cannot participate in the definition of the shard key, since the current version (4.4) of MongoDB does not support a 2dsphere indexed field to be a shard key or part of it.

[2]Broadcast operations may occur also for queries that include the shard key, depending on factors such as the data distribution on nodes and the query selectivity.

By setting the date field as the shard key, it is very likely that the documents will be distributed evenly among the shards, thus resulting in a load-balanced cluster. Chunks are unlikely to become *jumbo* (i.e., this refers to chunks that cannot be split, despite becoming too large), because the values of the documents that represent the temporal information ordinarily are of high cardinality. However, it is still disturbing that the spatial dimension cannot be used for sharding, since this would benefit queries with spatial constraints. Instead, such queries will inevitably be routed to all nodes that contain data that overlap with the temporal constraint of the query.

It should also be noted that an index is automatically constructed on the shard key in MongoDB. Thus, the choice of the date field for shard key results in two indexes: an index on the temporal information and the compound index on space time. During query processing, the query optimizer is responsible for deciding which index is going to be used.

4.1.3 Data locality and zones usage. The baseline solution can only guarantee data locality at the temporal level. This means that documents associated with locations that exist near in space will only be stored on the same node if they have similar timestamps. Still, neighboring temporal intervals (corresponding to chunks) may be stored in different nodes, due to the allocation of chunks to nodes. To improve this shortcoming, we can define zones on the sharding key (date), which will force neighboring temporal intervals to be stored on the same node, thus improving data locality with respect the temporal dimension.

In summary, sharding on the date field has the following two drawbacks: (i) spatio-temporal queries may be routed to nodes that fulfill the temporal range constraint but do not contain any data that satisfy the spatial constraint, and (ii) queries that are selective in the spatial dimension but refer to a large temporal interval are forwarded to many of nodes of the cluster.

4.2 Solution based on Hilbert SFC

4.2.1 Indexing. Instead of relying on the built-in spatial indexes, our approach is based on the 1D mapping of data by means of the Hilbert space-filling curve. Specifically, for each document, the 1D Hilbert value is determined given its longitude and latitude value, and then it is included as a new field (hilbertIndex) that stores this value (of type long), as shown in the example below.

```
{
    "_id": 1,
    "hilbertIndex": NumberLong(2345),
    "location": {
        "type": Point,
        "coordinates": [37.983810, 23.727539]
    },
    "date": ISODate("2018-09-12T12:15:17.777Z"),
    ...
}
```

Even though any 1D mapping could be employed, we select the 1D mapping values based on the Hilbert space-filling curve, as it has been shown to exhibit nice clustering and locality properties [14]. This means that two documents with values of hilbertIndex that differ slightly correspond to objects that are spatially close to each other.

4.2.2 Sharding. Given this new spatial field (hilbertIndex) and the date field, we set the shard key as {hilbertIndex, date}, thus imposing spatio-temporal partitioning of data to nodes. Consequently, the formation of the chunks is based both on spatial and temporal information and each of them contains documents that exist in specific spatial cells (1D values) for a certain time period. In case of spatio-temporal skewness in the data, chunks are unlikely to become *jumbo* because of the cardinality of the temporal field. The hilbertIndex field is more prone to skewness, as it consists of finite long values and some of them may appear with high frequency (i.e., correspond to frequently visited locations). Thus, when a chunk surpasses its configured size due to a unique hilbertIndex value, it is split on the temporal dimension. The two new chunks will refer to the same 2D region, covering different and non-overlapping time spans. The definition of {hilbertIndex, date} as the shard key creates by default a compound index for these fields on each shard. The index constitutes a spatio-temporal index that uses the spatial field first and then the temporal.

When querying such an index, not only the expected spatial and temporal constraints are included in the query as in the baseline approach, but also a constraint is added on the hilbertIndex field. This constraint practically includes the set of spatial cells that need to be examined, because they intersect with the query. More concretely, such a query has the following document representation in MongoDB:

```
{ $and: [
    { location: { $geoWithin: { $geometry: { type : Polygon,
        coordinates: [ [ [23.7397867, 37.9698929],
         [23.7492228,37.9698929], [23.7492228, 37.9761557],
        [23.7397867,37.9761557 ], [23.7397867, 37.9698929] ] ]
    } } } },
    {date: { $gte: ISODate("2019-04-18T12:15:14.002Z") } },
    {date: { $lte: ISODate("2019-04-24T07:34:43.777Z") } },
    { $or: [
        { hilbertIndex: { $gte: 7865, $lte: 12869 } },
        { hilbertIndex: { $gte: 13192, $lte: 13210 } },
       { hilbertIndex: { $in : [7794, 7799, 7856, 12911] } }
    ] }
] }
```

Similarly to the baseline approach, the spatio-temporal query uses the MongoDB *$geoWithin* operator on the field that stores the location as GeoJSON object. Also, the query specifies a specific temporal range using the *$gte* and *$lte* operators on the temporal field. However, in the Hilbert-based approach, an additional constraint is posed on the specific spatial cells. Consecutive values of cells are expressed as ranges, whereas non-consecutive cell values are included as individual values. For this purpose, an additional disjunctive operator (*$or*) is used in the query that contains *$gte* and *$lte* operands to represent the ranges, and an *$in* operator to include the individual cells.

4.2.3 Data locality and zones usage. The exploitation of both spatial and the temporal information of documents for sharding leads to a distribution of data to nodes that preserves the *spatio-temporal* data locality. It is highly probable for the documents that are both spatially and temporally near to be stored on the same node. However, as already explained, this is not guaranteed for data that belong to different chunks. This is because MongoDB does not guarantee (by default) for each of the shards to store chunks with continuous ranges.

Thus, data locality can be improved by defining zones. With zones, documents with consecutive Hilbert values (disregarding the temporal dimension) are likely to be placed in the same shard; this is applied by defining and assigning zones to shards, handling documents with specific ranges of Hilbert values. The

fact that the zones are defined only on the spatial information of the documents cannot guarantee temporal locality, but only for documents with nearby spatial locations. In contrast to the baseline solution, in this approach, the spatial dimension affects the number of the nodes that are to be accessed during query processing.

4.2.4 Zones configuration. In order to assign a pre-defined range of key values to a shard, a zone is specified by means of a range of shard key values or a prefix of shard key values. Zones are then assigned to certain nodes. This enables manual partitioning of data in a controlled manner. In order to achieve data locality regarding the spatio-temporal information of the documents, we define as many zones as the number of available shards, and assign one zone per shard.

The configuration of zones for the Hilbert-based approach is performed on the `hilbertIndex` field (whereas the shard key consists of the spatial and temporal field). Specifically, by using the *$bucketAuto* aggregation pipeline stage of MongoDB, we get the ranges of n buckets, where n is the number of shards. The boundaries of buckets are formed with the goal of even distribution of documents to buckets. The ranges of the zones contain documents that have a specific Hilbert value, without taking account its temporal part. In other words, the zones contain documents that are located in specific cells for the whole timespan in which they belong to. The resulting zones may not contain exactly the same number of documents due to spatial data skew, but we manage to preserve spatial locality.

For the baseline approach, the configuration of zones is performed on the `date` field that constitutes the shard key, using again the *$bucketAuto* aggregation pipeline stage. Each of the buckets, contains the same number of documents. Based on these ranges, the respective zones are created, and each one is assigned to a single shard.

5 EXPERIMENTAL EVALUATION

In this section, the results of the experimental evaluation using both real-life and synthetic data are presented.

5.1 Experimental Setup

Platform. All experiments were performed in Okeanos-Knossos[3], an IaaS platform which is built and supported by GRNET[4] for research purposes. Okeanos-Knossos is a cloud service that provides virtual computing and storage services to the Greek research and academic community. We have engaged from the cloud service the following resources: 17 virtual machines, 68 CPUs, 136GB RAM, and 2.00 TB disk space.

MongoDB deployment. We deployed MongoDB (v 4.0.12) on 17 virtual machines. From this set of nodes, 12 of the VMs were used as (primary) shards, 3 of them as configuration servers and the remaining 2 as query routers. Replica shards were not used since the availability feature which mainly relates to node failures is beyond the scope of this experimental study.

Each VM is equipped with 8GB RAM, x4 CPU cores and 30GB system disk, running Ubuntu 16.04.6 LTS operating system. The VMs that serve as shards have a mounted disk drive of 102GB size. The VMs that serve as query routers have a mounted disk drive of 145GB size. The offered attachable disk of Okeanos-Knossos platform is based on the Ceph[5] storage system, which is a distributed block level storage.

By default, MongoDB uses the WiredTiger storage engine that integrates compression for collections and indexes. The compression on the collections is achieved through block compression, supported through the snappy library. At the level of indexes, prefix compression is used.

Data sets. Two types of data sets are used for the experimental evaluation; real-life (R) and synthetic (S). The R set used in the experimental evaluation is a subset of a proprietary data set that is provided from a fleet management provider in Greece. The subset used in our experiments is 13.7GB in the form of CSV files, containing $15,210,901$ records in total. The data set contains trajectories of vehicles within Greece for a period of five months (July to November 2018). Particularly, each record constitutes a GPS trace of a specific vehicle and is comprised of 75 values. The values represent information about the vehicle, its position in space and time, the prevailing weather conditions, the road network and the nearest points of interest. The coordinates of the minimum bounding rectangle of the data set are: [(19.632533, 34.929233), (28.245285, 41.757797)]. We also use larger portions of the same data set for our scalability study in Section 5.4, by including more vehicles in the same spatio-temporal bounding box.

The S set is a synthetic spatio-temporal data set, which contains twice as many records as the R data set. It consists of two CSV files where each one contains 4 columns (`id`, `longitude`, `latitude` and `date`). The values of each column are generated at random within predefined ranges and following the uniform distribution. Its size is 1.6GB. The timespan of the synthetic data set is the half of the covering time period of the R data set (i.e., it spans 2.5 months) and covers spatially a smaller area than R. Specifically, the minimum bounding rectangle of the synthetic data set is approximately 1.54% of the minimum bounding rectangle area of the R set. The lower and upper coordinates of this rectangle are: [(23.3, 37.6), (24.3, 38.5)].

Queries. In order to assess the performance of the approaches, two categories of spatio-temporal queries are specified; those with small and big spatial constraint, respectively. The categories will be stated henceforth as Q^s (small) and Q^b (big) correspondingly. Each category contains 4 queries with increasing temporal constraint: Q_1^x covers 1 hour, Q_2^x 1 day, Q_3^x 1 week and Q_4^x covers 1 month, where $x \in \{s, b\}$. The queries do not overlap on the temporal dimension; instead, each one pertains to a discrete time span.

The spatial constraint of the small queries category is defined as a rectangle with the following lower and upper bounds; [(23.757495, 37.987295), (23.766958, 37.992997)]. The spatial rectangle covers 526km^2. The spatial constraint of the big queries category is also defined as a rectangle, approximately $2,603$ times larger than the covering area of the small queries category. Its lower and upper bound coordinates are [(23.606039, 38.023982), (24.032754, 38.353926)], covering $1,369,107$km^2.

Tables 2 and 3 report the number of retrieved documents for each small and big queries, respectively. Clearly, small queries are selective and retrieve relatively few documents. This corresponds to queries for constrained space and time dimensions, aiming at retrieval of very specific records. Instead, big queries, return large result sets, and correspond to analytic queries that retrieve

[3] https://okeanos-knossos.grnet.gr/
[4] https://grnet.gr/
[5] https://ceph.io/

	Number of retrieved documents			
Data set	Q_1^s	Q_2^s	Q_3^s	Q_4^s
R	2	34	877	3,829
S	3	22	239	783

Table 2: The results of small queries for the real and the synthetic data set

	Number of retrieved documents			
Data set	Q_1^b	Q_2^b	Q_3^b	Q_4^b
R	580	5,640	113,890	431,788
S	2,575	61,193	608,685	1,891,291

Table 3: The results of big queries for the real and the synthetic data set

large portions of the data set in order to perform some large-scale data analysis task.

Metrics. The evaluation of the performance of the approaches is based on the following metrics;

- *Average execution time*: corresponds to the execution time required for processing the query and returning the query result (averaged over queries).
- *Documents examined*: corresponds to the maximum number of documents that were accessed on any node during query processing. This indicates the number of documents that need to be fetched from disk during query processing. We use the maximum as it corresponds to the highest cost induced on any node, so it affects the execution time.
- *Keys examined*: shows the maximum number of keys examined in the underlying index over all nodes, in order to find the necessary documents on disk. This indicates the cost paid during query processing induced by accessing the index.
- *Nodes*: reports the number of nodes that were accessed during the execution of a query. This corresponds the subset of nodes that participated in query processing.

Methodology. Based on the description of the subsections 4.1 and 4.2, we evaluate the following individual approaches which cover variants both for indexing and partitioning:

- *bslST*: sharding based on time, and local indexing on shards using compound index (`location`, `date`), where location is based on 2dsphere index.
- *bslTS*: sharding based on time, and local indexing on shards using compound index (`date`, `location`), where location is based on 2dsphere index.
- *hil*: range sharding based on Hilbert curve in 2D (with 13-bit precision) and on time. The applied Hilbert curve covers the entire globe. For local indexing on shards, a compound index (`hilbertIndex`, `date`) is used for each node.
- *hil**: range sharding based on Hilbert curve in 2D (with 13-bit precision) and on time. The applied Hilbert curve is limited to the spatial region of the data set. For local indexing on shards, a compound index (`hilbertIndex`, `date`) is used for each node.

With the *bsl* term we will refer to the sharding of the bslST and bslTS approaches which is common for both of them. These approaches differ only on their indexing part. Furthermore, the hil method competes on equal terms bslST and bslTS approaches, by taking account of the same spatial extent (whole world) and by using 26 bits for storing the geospatial information of each document on the respective indexes. We also tried hil* to investigate if the use of a curve with the same number of bits but on restricted spatial extent has any notable effect on performance.

The evaluation methodology followed in this experimental study is described as follows. Each of the queries of the two categories are executed 30 times, so as to ensure that the measurements of the performance time of queries are in warm state (where indexes have been already loaded in-memory). The execution time is calculated by averaging the last 10 executions of each query.

When switching from one approach to the other, MongoDB is re-installed from scratch so as to have a new deployment which will contain only one sharded collection. For each approach, data loading takes place. Both of the query routers are exploited for this purpose and perform bulk insertion.

The approaches are tested under range sharding with different settings: (i) with the default formation of the chunks, without intervening on the distribution of the data over the shards (Section 5.2, Figs. 5–8), and (ii) with the definition of zones where data is grouped to shards given some pre-determined ranges (Section 5.3, Figs. 9–12).

5.2 Evaluation of Approaches

***hil* vs. *bsl*.** For both of the baseline variants (bslST and bslTS), the number of nodes to which a query is routed increases when the temporal constraint of a spatio-temporal query is increasing, regardless of its spatial extent (Fig. 5c, 6c, 7c, 8c). This is not effective for big queries that cover a large spatial area and refer to a short time period (such as Q_1^b and Q_2^b in Fig. 6d and Fig. 8d). In that case, a small number of nodes are burdened with search operations, as a result of the multiple 1D mapping values that represent the area. This is reflected in Figs. 6a, 6b, 8a and 8b where the maximum number of examined keys and documents are many more than in hil. The same applies for the Q_3^b query, served by 2 and 6 nodes in the R and S set respectively, fewer than the exploited nodes of hil method. The query Q_4^b uses in the baseline approaches 3 nodes more than in hil for both sets, with more maximum examined keys and documents. In summary, hil outperforms the baseline methods in terms of execution time in the case of big queries.

Concerning the small queries, Q_1^s and Q_2^s perform better for hil in R set, since the maximum examined keys are fewer than in bslST and bslTS (Fig. 5a). The same applies for the S set in terms of the performance, but for the Q_2, a few more documents are examined in hil than bslST (Fig. 7a). Still, Q_2^s performs a little better in *hil* than bslST, as 1 node is used, thus without having the small overhead of merging the results from the shards. The opposite happens for the maximum examined documents (Fig. 5b and 7b) for the same queries on the two sets, but the differences are smaller than the respective cases of the examined keys. For queries Q_3^s and Q_4^s, bslST outperforms hil. This happens because more nodes contribute to the execution of the queries. Since the queries are spatially small, fewer nodes are used in the hil method.

hil* vs. *hil*.** The performance of hil and hil* methods differ in favor of hil for the big queries in both sets (Fig. 6d and 8d). The same does not apply for the small queries (Fig. 5d and 7d), excluding Q_1 for both sets and Q_2 for S set. In practice, hil* uses higher precision in the spatial domain than hil, having more

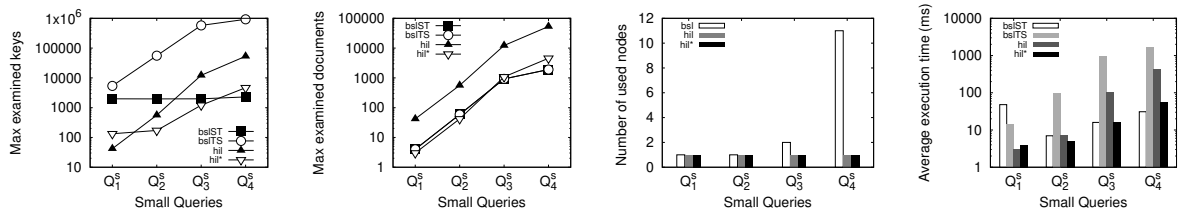


Figure 5: Default sharding ranges: Results for small queries and real (*R*) data

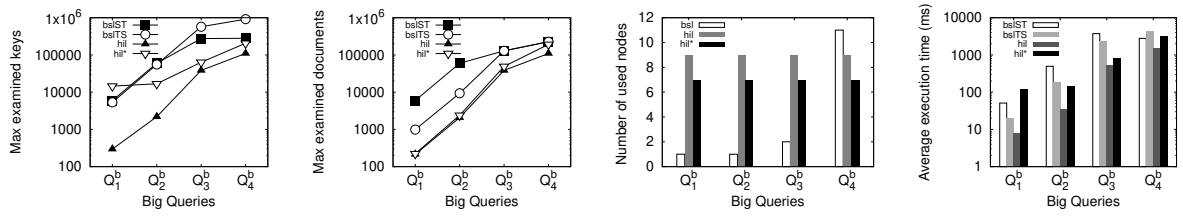


Figure 6: Default sharding ranges: Results for big queries and real (*R*) data

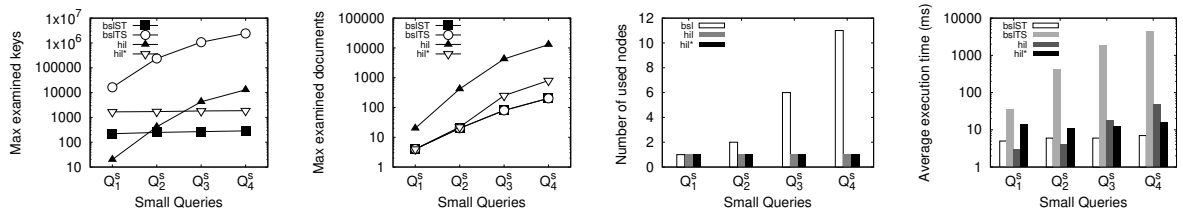


Figure 7: Default sharding ranges: Results for small queries and synthetic (*S*) data

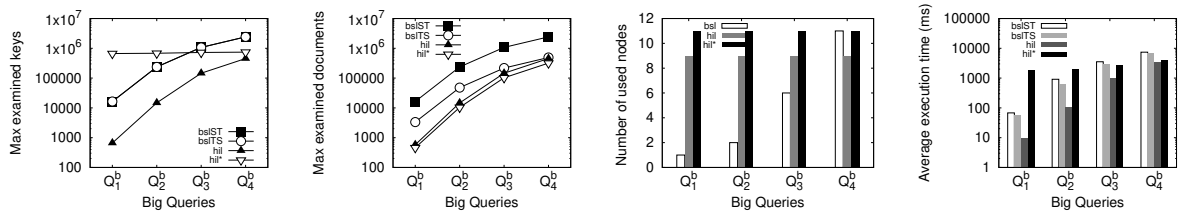


Figure 8: Default sharding ranges: Results for big queries and synthetic (*S*) data

indexed values on its spatial part. This results to a compound index with more buckets but with fewer elements on each bucket. It is expected, that such an index will perform better for spatio-temporal queries that cover a small area, than an index with fewer values on the spatial part. The difference in the performance is greater when the temporal dimension is increasing, since many of the buckets of the hil* that fulfill the spatial criteria will not be accessed, because of their temporal boundaries, thus resulting to fewer examined documents.

Discussion. In general terms, our experiments with the default distribution settings validate our intuition that the integration of the spatial information on the sharding offers performance gain. This is especially evident for big queries where hil outperforms bsl. In the case of small queries, bsl performs better but at the expense of using more nodes for query execution due to the lack of data locality. This is going to have a negative effect in a real system that processes thousands of queries at the same time, since it would require that all nodes need to participate in the execution of each query, which is not scalable.

5.3 Evaluation of Approaches with Zones

In this set of experiments, we only use hil (not hil*) since we did not observe significant performance improvements. Many of the experimental observations of Section 5.2 stand also for the case of grouping data via zones. For instance, for all of the big queries (Figs. 10d and 12d), hil outperforms both bslST and bslTS, because the maximum number of examined documents is smaller. Moreover, the required execution time of small queries

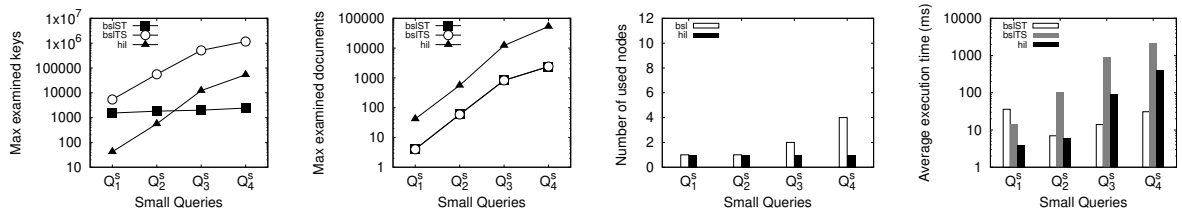


Figure 9: Zone ranges: Results for small queries and real (*R*) data

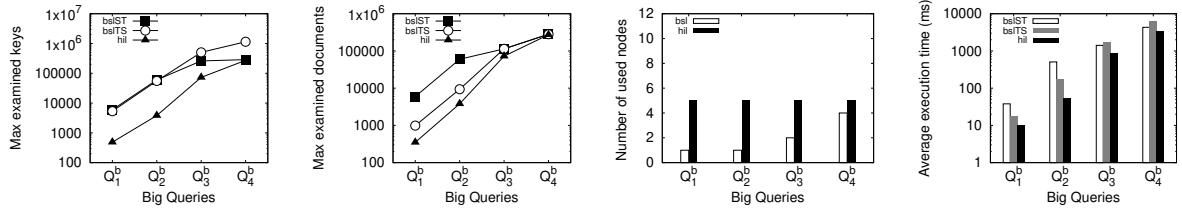


Figure 10: Zone ranges: Results for big queries and real (*R*) data

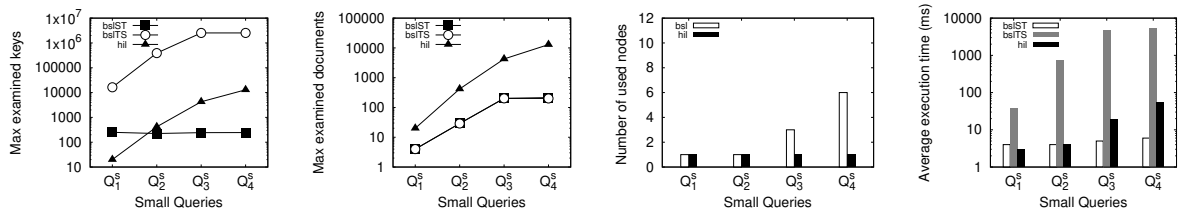


Figure 11: Zone ranges: Results for small queries and synthetic (*S*) data

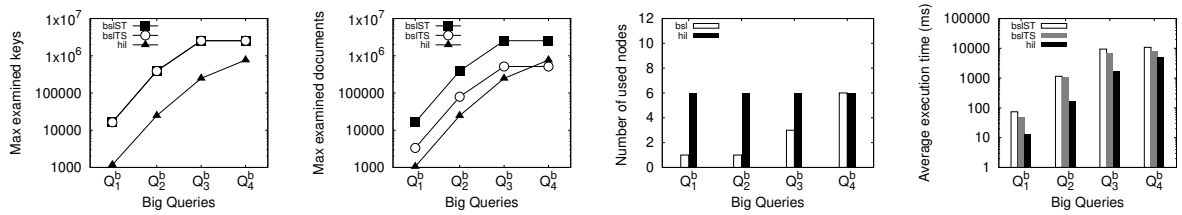


Figure 12: Zone ranges: Results for big queries and synthetic (*S*) data

with a large timespan (Q_3 and Q_4) is smaller for the bslST than hil, since more nodes are exploited, resulting to a smaller number of maximum examined keys and documents for both sets (Figs. 9a, 9b, 11a and 11b).

It should be underlined that in the cases in which more than two node were exploited for the performance of a query in the experiments of Section 5.2, their corresponding application over of the pre-defined zones use fewer nodes. This is expected because of the grouping of zones on shards. For those cases, the query performance drops to a small degree. Additionally, better data locality is offered via zones as data is moved to specific shards, adhering to the specified ranges.

Discussion. The usage of zones for each of the approaches confirms that data locality is better than in the case of default distribution of the documents among the shards. This is demonstrated by the fact that fewer (or, in some cases, the same number) of nodes take part in the processing compared to the case of no zones.

In the baseline approaches, the Q_4 big queries in both R and S data sets consume more time for their execution, as fewer nodes are involved in query processing. The performance degradation is also noticed for the Q_2^b and Q_3^b in the S set, whereas the respective queries gain performance in the R data set. Query Q_1^b gains slightly in performance in all cases. Regarding the small queries, in the bslST approach, query execution remains practically the same. In the bslTS approach, the performance of Q_4^s gets worse for both R and S data sets. The same applies for Q_3^s and Q_2^s, but in

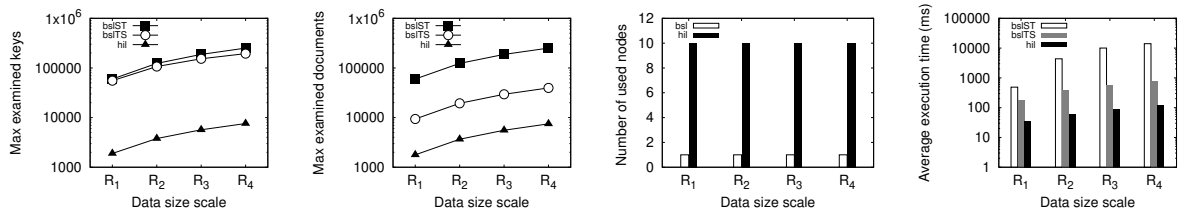


Figure 13: Scalability study for default sharding ranges: Results for Q_2^b query on real data

the S data set. The performance of the remaining queries remains the same.

For the Hilbert-based approach, the usage of zones affects only the execution of Q_2^b, Q_3^b and Q_4^b big queries, increasing their execution time as a smaller number of nodes are involved in query processing. An extra number of keys and documents have to be searched. This is validated in both R and S data sets. The remaining queries have similar performance, except for small queries Q_3^s and Q_4^s in the R data set. Their execution time is slightly decreased despite that the same number of nodes (namely one node) is still exploited with zones. These queries have been favored due to the change in the distribution of data by using zones.

5.4 Scalability Study

In this experiment, we study the scalability of our approach for larger data sets. For this purpose, we use larger portions of the real R data set. In particular, we use R_1 to denote the data set used so far, whereas R_2, R_3, and R_4 correspond to larger data sets, by a scale factor of x2, x3, and x4 respectively. Table 4 reports the size (in GB) and number of documents for each data set. Notice that we keep the same spatio-temporal bounding box for the data set, and we obtain larger instances of the data set by adding data from more vehicles. We select query Q_2^b in order to study its performance when the size of the data set is increased. Also, Table 5 reports the number of results for Q_2^b for the different data sets ($R_1 - R_4$), showing that the same query retrieves more results as the scale factor of the data set increases.

	Data sets with scale factor			
Data set info	R_1	R_2	R_3	R_4
Size (GB)	40.8	83.87	127.27	171.59
#documents (M)	15.2	31.4	47.7	63.9

Table 4: Information for instances $R_1 - R_4$ of the real data set for different scale factor

	Data sets with scale factor			
Query	R_1	R_2	R_3	R_4
Q_2^b	5,640	11,792	17,840	23,854

Table 5: Number of results for query Q_2^b per scale factor of the real data set

Fig. 13 demonstrates that our approach scales gracefully as the size of the data is increased. This is particularly evident when considering the execution time in Fig. 13d. Perhaps more importantly, the gain of hil over the baseline methods increases with the size of the data, which favors the performance of hil for larger data collections. Figs. 13a and 13b show that hil needs to access 1-2 orders of magnitude fewer documents and 2 orders of magnitude fewer keys respectively. When comparing the two baselines, bslTS performs better than bslST, since fewer documents are examined in the query's refinement phase. Recall that query Q_2^b is selective on its temporal dimension (covering only one day) and thus it is expected that bslTS will perform better than bslST.

Discussion. This experiment demonstrates that the proposed approach (hil) sustains its benefits over the baseline methods when the size of the underlying data set is increased. This indicates that our approach is scalable with data size.

6 CONCLUSIONS

In this paper, we provide an in-depth study of spatio-temporal query execution in NoSQL stores, focusing on MongoDB due to its popularity and support for spatial data. Our study indicates that existing NoSQL stores do not natively support spatio-temporal data, despite the ever-increasing number of applications that produce massive spatio-temporal data daily. We also show that a baseline approach based on built-in spatial indexes leads to suboptimal performance in several cases. More importantly, we investigate on the underlying reasons for this impact on performance, and we elaborate on indexing and sharding techniques. Furthermore, we propose an alternative approach which exploits the Hilbert curve to map data to 1D values, which are then: (a) indexed using a standard B-tree, and (b) used for partitioning the data in a way that preserves spatio-temporal data locality in shards. We conclude that this has an effect on the number of nodes storing data that participate in query execution. Our extensive experiments using both real-life and synthetic data in a cluster of nodes support our conclusions.

With respect to future work, we believe that big data developers that work on spatio-temporal data will find interest in our work, especially towards optimizing the performance of their applications. Also, we expect that future releases on NoSQL stores will provide support for spatio-temporal data, therefore our work is a small contribution in this direction. As a result, we intend to extend our approach and investigate other methods for indexing and partitioning spatio-temporal data in distributed NoSQL systems. Also, extending our work towards supporting more complex data types (polylines and polygons) is of interest. Last, but not least, we would like to expand our study using a workload of queries, and propose an adaptive, workload-aware mechanism for indexing and partitioning.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780754 (Track&Know project), and the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT) under grant agreement No HFRI-FM17-81 (Chorologos project).

REFERENCES

[1] Dominik Bartoszewski, Adam Piórkowski, and Michal Lupa. 2019. The Comparison of Processing Efficiency of Spatial Data for PostGIS and MongoDB Databases. In *Beyond Databases, Architectures and Structures. Paving the Road to Smart Data Processing and Analysis - 15th International Conference, BDAS 2019, Ustroń, Poland, May 28-31, 2019, Proceedings (Communications in Computer and Information Science, Vol. 1018)*, Stanislaw Kozielski, Dariusz Mrozek, Pawel Kasprowski, Bozena Malysiak-Mrozek, and Daniel Kostrzewa (Eds.). Springer, 291–302.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, Hector Garcia-Molina and H. V. Jagadish (Eds.). ACM Press, 322–331.

[3] Norbert Beckmann and Bernhard Seeger. 2009. A revised r*-tree in comparison with related index structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 799–812.

[4] Rick Cattell. 2010. Scalable SQL and NoSQL data stores. *SIGMOD Rec.* 39, 4 (2010), 12–27.

[5] Kristina Chodorow. 2013. *MongoDB: The Definitive Guide.* O'Reilly Media, Inc.

[6] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.

[7] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A Survey on NoSQL Stores. *ACM Comput. Surv.* 51, 2 (2018), 40:1–40:43.

[8] Cédric du Mouza, Witold Litwin, and Philippe Rigaux. 2009. Large-scale indexing of spatial data in distributed repositories: the SD-Rtree. *VLDB J.* 18, 4 (2009), 933–958.

[9] Miaoran Duan and Gang Chen. 2015. Assessment of MongoDB's spatial retrieval performance. In *23rd International Conference on Geoinformatics, Geoinformatics 2015, Wuhan, China, June 19-21, 2015*, Shixiong Hu (Ed.). IEEE, 1–4.

[10] Xuefeng Guan, Cheng Bo, Zhenqiang Li, and Yaojin Yu. 2017. ST-hash: An efficient spatiotemporal index for massive trajectory data in a NoSQL database. In *25th International Conference on Geoinformatics, Geoinformatics 2017, Buffalo, NY, USA, August 2-4, 2017*. IEEE, 1–7.

[11] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, Beatrice Yormark (Ed.). ACM Press, 47–57.

[12] Anand Padmanabha Iyer and Ion Stoica. 2017. A scalable distributed spatial index for the internet-of-things. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 548–560.

[13] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, and Dimosthenis Anagnostopoulos. 2019. Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data. In *Proceedings of the Workshops of the EDBT/ICDT 2019 Joint Conference, EDBT/ICDT 2019, Lisbon, Portugal, March 26, 2019 (CEUR Workshop Proceedings, Vol. 2322)*, Paolo Papotti (Ed.). CEUR-WS.org.

[14] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. 2001. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE Trans. Knowl. Data Eng.* 13, 1 (2001), 124–141.

[15] Georgios M. Santipantakis, Apostolos Glenis, Kostas Patroumpas, Akrivi Vlachou, Christos Doulkeridis, George A. Vouros, Nikos Pelekis, and Yannis Theodoridis. 2020. SPARTAN: Semantic integration of big spatio-temporal data from streaming and archival sources. *Future Gener. Comput. Syst.* 110 (2020), 540–555.

[16] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, Peter M. Stocker, William Kent, and Peter Hammersley (Eds.). Morgan Kaufmann, 507–518.

[17] Chaowei Yang, Keith C. Clarke, Shashi Shekhar, and C. Vincent Tao. 2020. Big Spatiotemporal Data Analytics: a research and innovation frontier. *Int. J. Geogr. Inf. Sci.* 34, 6 (2020), 1075–1088.

A ADDITIONAL EXPERIMENTAL RESULTS

We provide additional details on the empirical evaluation, including the bulk loading technique that we use, details on index usage, as well as information of query distribution to nodes and index size.

A.1 Data Loading

Data Loading. Both of the nodes that act as query routers, store half of the CSV files of the data sets on their disks. Data loading to the MongoDB store is carried out by accessing the CSV files record-by-record and converting them directly to documents (binary JSON - BSON format). The conversion process does include the formation of a GeoJSON object as a value of the `location` field based on the longitude and latitude columns of the CSV datasource files. It also includes the addition of the fields with its respective values for all of the rest columns of the CSV files, and the addition of the `_id` field which is handled automatically by the MongoDB client driver. Note that for the Hilbert-based approaches, a further action takes place, related to the calculation of the 1D value. When a document is formed, it is added to a list. The list is used for bulk insertion of documents for performance reasons. The insertion is triggered after a specific batch of elements has been placed in the list, and we use 15K documents as batch size in our experiments.

The size of the R and S set is a marginally smaller in the bsl method than hil and hil* (Table 6), since its documents do not integrate the `hilbertIndex` field, as hil and hil* do. Also, the sizes of the methods in R set are much larger than S set because they incorporate much more information due to the extra fields.

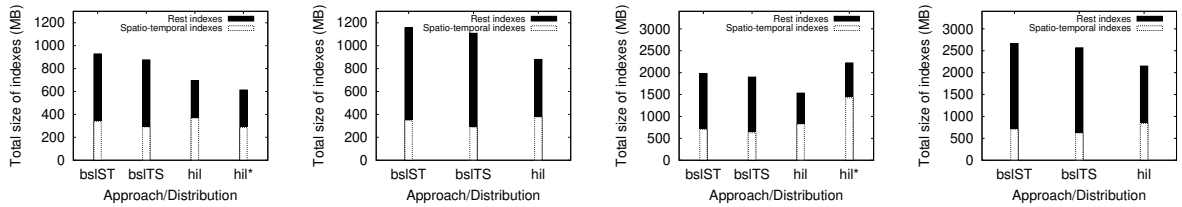
	Approach	
Data set	*bsl*	*hil(*)*
R	40.54	40.8
S	3.62	4.13

Table 6: Data size of real and synthetic data set in MongoDB (in GB)

A.2 Querying the Data

The bslST and the bslTS approaches use as a sharding key the `date` field which results to an additional index per node, apart from the compound index. As a result, there are some cases where the query optimizer chooses to process the spatio-temporal query at hand using the index on `date`, rather than using the compound index. This occurs only in the bslST approach, whereas in the bslTS approach all queries are processed using the compound index. Table 7 reports which index was used during query processing for the bslST approach for all queries and both data sets.

Querying under the hil and hil* approaches, requires the determination the cell of the indexes that cover the queries' spatial extent. Table 8 shows the average time execution of the Hilbert algorithm for the identification of the cell indexes. When applying the hil* methodology, it is reasonable that the algorithm requires more time than hil for specifying the 1D values, as the total searching space is limited to a smaller surface, resulting to increased precision. Furthermore, the execution time in the *S* data set is increased when comparing the respective approaches (except for Q^s for hil) to the *R* set, since the data exists in a smaller 2D space which increases the precision. This adds a burden to the algorithm's operation for identifying the 1D values. The figures that follow in the next subsections, showing query execution time, do not include the time for the determination of the 1D values through the space filling curve.



(a) R set with default distribution **(b) R set with zones** **(c) S set with default distribution** **(d) S set with zones**

Figure 14: Total size of indexes for R and S set with default distribution settings and zones

Distribution	Data set	Q^x	Q^x_1	Q^x_2	Q^x_3	Q^x_4
Default	R	Q^s	●	●	●	●
		Q^b	○	○	●	◐[a]
	S	Q^s	●	●	●	●
		Q^b	○	○	○	○
Zones	R	Q^s	●	●	●	●
		Q^b	○	○	●	◐[b]
	S	Q^s	●	●	●	◐[c]
		Q^b	○	○	○	○

Table 7: Usage of indexes for the bslST approach

●The used nodes exploit the compound index, ○the used nodes exploit the date index, ◐mixed usage of the two indexes among the used nodes - the most delayed node uses the compound index

[a] 3 of the 11, [b] 3 of the 4 and [c] 5 of the 6 used nodes exploit the compound index

	Methodology			
	hil		*hil**	
Data set	Q^s	Q^b	Q^s	Q^b
R	0.05	0.2	0.1	1.8
S	0.05	0.3	0.6	7.6

Table 8: Average time of Hilbert algorithm performance in ms, for finding which 1D values should be searched on the index

A.3 Index Size

The baseline methods have a few more main memory resources requirements than hil for maintaining the indexes among the shards. This is indicated in Figs 14a, 14b, 14c and 14d. Except for the spatio-temporal indexes, bslST and bslTS maintain additionally two indexes which are created by default; one for the *_id* field of every document, and the other for the *date* field, as it acts as the sharding key. Whereas, hil and hil* methods maintain as an additional index apart from the spatio-temporal, only the one that is used for the _id field. Their spatio-temporal indexes preexist by default due to the fact that its fields constitute the sharding key. This, favors especially the hil method, since in all cases, less memory is required for handling all the indexes than bslST and bslTS.

For each of the methods, both in R and S set, the total size of indexes among the shards increases when transiting from the default data distribution to zone ranges. The change in size is affected by the indexes' size that handle the _id field. The insertion of the documents in MongoDB database, takes place under the default distribution settings, and the documents with similar construction time have a larger common prefix part on the _id field. Documents created in similar timestamps that get inserted in the same shard contribute to the occupation of less main memory by the _id index, since MongoDB uses prefix compression. With the definition of zones, the documents that have been already stored in the shards, are shuffled around the cluster, resulting to shards with documents that have a smaller common part. This makes the compression less effective and thus the sizes of _id indexes are increased.

Moreover, the total size of the spatio-temporal indexes does not change significantly when shifting from the default data distribution state to zone ranges. The indexes have approximately the same size. The same applies for the date indexes that exist in bslST and bslTS methods; their size remains approximately the same for both default distribution and zone ranges.

Comparing the hil and hil* total size of indexes, it is noticed that in R set less main memory is required for hil* method (Fig. 14a), whereas the opposite is true for the S set (Fig. 14c). This is reasonable because for the S set, the cardinality of the 1D values for hil* is much greater than hil method when contrasting the same methods for the R set. In R set, many are the documents that have the same 1D value as they are spatially skewed. The values are grouped to buckets, covering each one a specific time period. This saves memory especially when having smaller buckets that still cover a specific time period; this is achieved through hil*. On the other side, in S set, hil* is less efficient in terms of memory occupation than hil because the 1D hilbert values that are handled, are many more as the data is distributed uniformly all over their minimum bound box. This makes the compression less effective either, thus increasing the size of the hil* index.

Industrial Paper

Production Experiences from Computation Reuse at Microsoft

Alekh Jindal
Gray Systems Lab
Microsoft
alekh.jindal@microsoft.com

Shi Qiao
Azure Data
Microsoft
shqiao@microsoft.com

Hiren Patel
Azure Data
Microsoft
hirenp@microsoft.com

Abhishek Roy
Gray Systems Lab
Microsoft
abhishek.roy@microsoft.com

Jyoti Leeka
Gray Systems Lab
Microsoft
jyoti.leeka@microsoft.com

Brandon Haynes
Gray Systems Lab
Microsoft
brandon.haynes@microsoft.com

ABSTRACT

Massive data processing infrastructures are commonplace in modern data-driven enterprises. They facilitate data engineers in building scalable data pipelines over shared datasets. Unfortunately, data engineers often end up building pipelines that have portions of their computations common across other pipelines over the same set of shared datasets. Consolidating these data pipelines is therefore crucial for eliminating redundancies and improving production efficiency, thus saving significant operational costs. We had built CloudViews for automatic computation reuse in Cosmos big data workloads at Microsoft. CloudViews added a feedback loop in the SCOPE query engine to learn from past workloads and opportunistically materialize and reuse common computations as part of query processing in future SCOPE jobs — all completely automatic and transparent to the users.

In this paper, we describe our production experiences with CloudViews. We first describe the data preparation process in Cosmos and show how computation reuse naturally augments this process. This is because computation reuse prepares data further into more shareable datasets that can improve the performance and efficiency of subsequent processing. We then discuss the usage and impact of CloudViews on our production clusters and describe many of the operational challenges that we have faced so far. Results from our current production deployment over a two month window show that the cumulative latency of jobs improved by 34%, with a median improvement of 15%, and the total processing time reduced by 37%, indicating better customer experience and lower operational costs for these workloads.

1 INTRODUCTION

Modern data-driven enterprises rely on large-scale data processing infrastructures for deriving business insights. As a result, over the last decade, a plethora of tools have been developed that have democratized scalable data processing for data engineers and data scientists. Examples include MapReduce [14], Spark [5], Hive [4], Presto [34], BigQuery [19], Athena [6], and SCOPE [9, 47]. However, the large-scale data processing infrastructures also incur massive operational costs and therefore improving their efficiency becomes very important in production. Interestingly, easy access to large scale infrastructure often leads data engineers to quickly build data processing pipelines that later end up having portions of computations repeated across one another. Furthermore, these redundancies are hard to discover in large enterprises with thousands of developers spread across different business units. Thus, we need automated tools to consolidate these data pipelines and improve operational efficiency, without impeding the developer productivity in quickly going all the way from raw data to actionable insights.

We see two key challenges when considering approaches for automatic compute reuse. First, it is very tedious to automatically detect the common computations in large volumes of complex analytical queries that are declarative and often include custom user code. Our analysis over a large window of production workloads, consisting of 67 million jobs and 4.3 billion sub-computations (referred to as *query subexpressions*), show that more than 75% of the query subexpressions are repeated. However, not all of the common computations are going to be viable candidates for reuse, e.g., due to very large storage overheads. Therefore, carefully detecting and selecting the common computations for reuse is a challenge. And second, there is a shift towards serverless query processing infrastructures [6, 9, 19], also sometimes referred to as job services, where users simply submit their declarative SQL-like queries without provisioning any infrastructure. As a result, users do not have any spare offline cycles for materializing the common computations before running their actual data analysis. In fact, users want to get started with their analysis quickly and they would rather have any optimizations applied in an online and adaptive manner.

We also see two key opportunities. First, there is a presence of large volumes of shared datasets in enterprises. This is because raw logs and telemetry coming in from various products are extracted and preprocessed into a shape and form that could be easily consumed by thousands of downstream developers. The resulting shared datasets are written once and read many times. Furthermore, they get regenerated periodically without requiring any fine-grained updates. As a result, the shared computations also do not need to be maintained with updates. And second, computation reuse holds the promise of significantly improving both the job performance and the operational efficiency, which are crucial in speeding up the time to insights and to enable developers to do more with the same set of resources (that are likely to have longer procurement cycles).

To address the above computation reuse problem, we had built CloudViews for automatic computation reuse in big data workloads at Microsoft [26]. Specifically, CloudViews optimizes the SCOPE query workloads in Cosmos big data analytics platform, that is used in various business, such as Bing, Windows, Office, Xbox, etc., across the whole of Microsoft. CloudViews identifies the common computations (query subexpressions) across

different SCOPE jobs, selects the ones that could lead to more efficiency, materializes them as part of query processing with minimal overhead, reuses them in future jobs — all completely automatic and transparent for the users. In contrast to prior works on materialized views [20] and multi query optimizations [37], CloudViews materializes common query subexpressions as part of query processing, i.e., does not require any offline cycles, and automatically replaces older materialized views with newer ones when the shared datasets are bulk updated, i.e., no update maintenance. Production experiences with Cosmos showed that bulk updates are a significant part of the workload, incremental updates if required can be handled using techniques proposed by Wang, et al.[42]. CloudViews considers only the same logical query subexpressions (with some normalization) for reuse, i.e., it does not consider view containment [21]. Although this is helpful in getting more accurate statistics for estimating the utility and cost of reusing different candidate subexpressions. CloudViews uses these estimates to select the set of subexpressions to materialize such that they provide the maximize reuse within a given storage budget [24]. We plan to add query containment to CloudViews as part of future work.

In this paper, we describe our production experiences with CloudViews. We delve deeper into the data preparation process in Cosmos, referred to as *data cooking*, and discuss how computation reuse naturally augments data cooking by creating more shareable datasets that can boost the performance and efficiency of further downstream processing. While we previously showed the impact of CloudViews on TPC-DS workloads and a small set of production queries in pre-production environment, we now analyze the usage of CloudViews in production and compare the impact along several performance metrics. Apart from performance improvements, CloudViews also introduced an automated feedback loop in the SCOPE query engine, i.e., moving towards a self-tuning model [25, 26]. As a result, there were several operational challenges that we have faced and valuable lessons learned along the way. We discuss these challenges and also reflect back on our journey from research to product.

In summary, we make the following key contributions in this paper:

(1) We provide a detailed description and analysis of data cooking in Cosmos and reason about how compute reuse with CloudViews helps augment the data cooking process. We also discuss the impact of various design decisions that were made in CloudViews. (Section 2)
(2) We present an analysis of the usage and impact of CloudViews on our production workloads. Results from our production deployment show that over a two month window the cumulative latency of jobs improved by 34%, the total processing time reduced by 37%, and the total number of containers used for processing dropped by 36%. We also highlight some of the other non-obvious implications, including smaller inputs (by 36%), less data read (by 39%), and shorter queue lengths (by 13%). (Section 3)
(3) We discuss several operations challenges faced, including challenges in view selection when considering job scheduling, correctness guarantees, dependencies with other components, customer on-boarding, customer expectations, and quantifying the impact over constantly changing workloads. (Section 4)
(4) Finally, we reflect back on the journey from taking a research prototype to production, describe how the ability to create derived data as part of query processing is a powerful mechanism in general, and present insights for many of the open opportunities that we see going forward (Section 5).

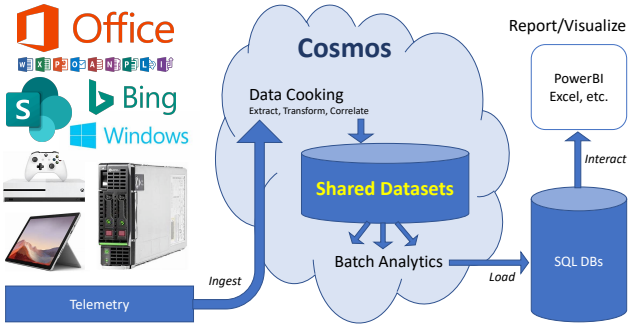


Figure 1: Illustrating typical data cooking in Cosmos.

2 DATA COOKING

Cosmos powers the internal big data analytics at Microsoft, with a massive infrastructure consisting of multiple 50k+ node analytics clusters [35], and processing declarative analytical queries using the SCOPE query engine. Cosmos runs hundreds of thousands of batch SCOPE jobs per day, consuming millions of containers and crunching over several petabytes of data generated per day. Almost 80% of the SCOPE workloads are recurring in nature [28], i.e., similar jobs templates are executed periodically at regular intervals over new data sets and parameters. Furthermore, SCOPE jobs have dependencies across each other. In fact, 80% of the jobs depend on at least one other job, while 68% have dependencies in a recurring fashion [12]. This is because of the presence of large volumes of shared data sets in Cosmos. Below we describe this enterprise pattern in more detail.

2.1 An Enterprise Pattern

Figure 1 illustrates the typical data cooking pipeline in Cosmos, where the raw telemetry data from different Microsoft products and services are ingested into the Cosmos store. Thereafter, the data cooking process extracts the structured data, transforms it into better representation, and correlates across multiple sources. The resulting shared datasets are generated periodically and consumed in multiple downstream batch-oriented analytics. Aggregated data is then loaded into interactive query processing systems like SQL databases for interactive analysis, reporting, and visualization using tools such as PowerBI or Excel. We can see that data cooking and the ability to collect and process large volumes of shared datasets across various developers and even business units is at the core of the above pipeline.

Figure 2 shows cumulative distributions of shared data sets and their consumers in five of our production clusters over a one-week window. We can see that more than half of the datasets are shared across multiple distinct consumers. Furthermore, several datasets are consumed tens to hundreds of times, with few getting reused thousands of times as well. Cluster1 in particular sees more shared data sets since that feeds into the Asimov platform that *implements a new mechanism for user feedback, allow for the testing of new features to gauge user acceptance, track bugs, and easily roll out new functionality and fixes* [33]. In fact, 10% of the inputs on this cluster get reused by more than 16 downstream consumers. For other clusters, 10% of the inputs are consumed

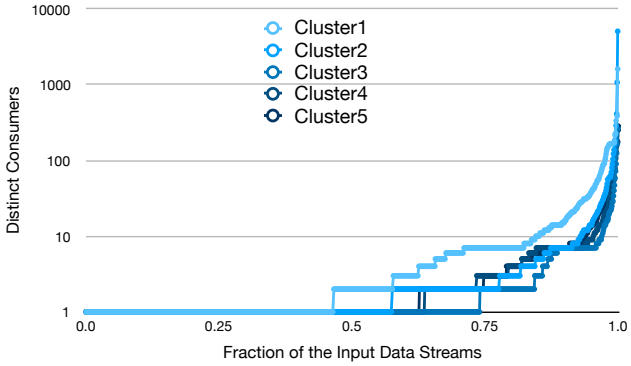


Figure 2: Shared data sets in five production clusters.

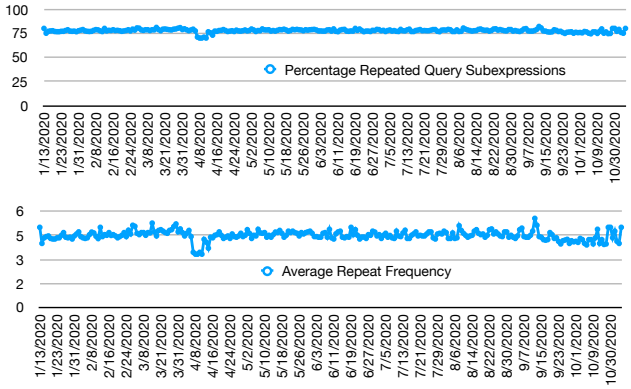
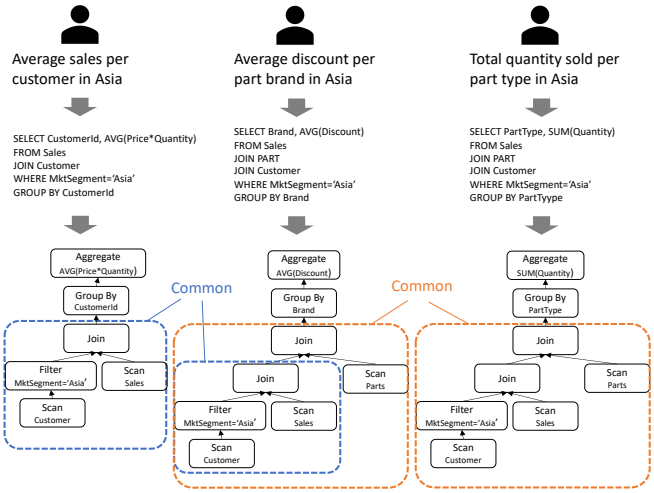
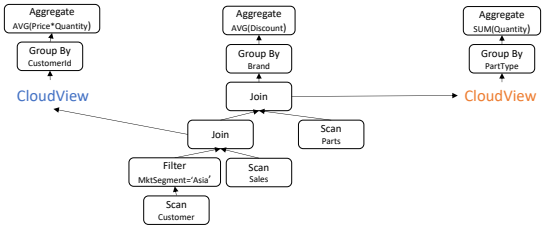


Figure 3: Overlaps in production clusters.



(a) Query plans with common computations across different users.



(b) Modified query plans with computation reuse across users.

Figure 4: Illustrating computation reuse across three analysts working on the same datasets.

by 7 or more downstream consumers. Thus, we see that shared data sets are prevalent in Cosmos.

2.2 Augmented Data Cooking

We saw that enterprise data analytics involves cooking massive volumes of data into a form that is consumable by several users. The shared datasets are therefore also a natural habitat for shared data analytics. Furthermore, it turns out that computation reuse is a natural problem in shared data analytics since often there are same sets of transformations that are applied repeatedly by analysts over the same shared datasets. Ideally, these shared computations could be captured in the cooking process itself. However, that is hard practically due to the lack of visibility into the downstream analytics (e.g., by different teams or business units consuming the shared datasets), complexity of SCOPE queries where it is non-trivial to identify the shared computations (portions of declarative large queries, including user defined functions, that may end up getting compiled to the same query sub-plan), or even due to the evolving nature of the analytics (new reports or dashboards being created). In a way, computation reuse can *augment* the handcrafted data cooking process by further fine-tuning the shared datasets with reusable views that are automatically identified, adapted, and created just in time, based on the workloads. Thus, we argue for computation reuse to be a first-class citizen in conjunction with data cooking for shared data analytics.

We analyzed the overlaps in our workloads in five of our production clusters over a 10-month window (January–October, 2020). Overall, there were 67 million jobs consisting of 4.3 billion query subexpressions from 2.5K users and 776 virtual clusters[1]. Figure 3 shows that more than 75% of query subexpressions are consistently overlapping over the 10-month window. Furthermore, the average repeat frequency consistently hovers around 5, indicating that materializing and reusing could be helpful for many of these overlapping computations. We refer interested readers to [26] for more fine-grained analysis of overlaps over a single day window.

In summary, computation reuse can fill the gaps in data cooking, and we see significant opportunities for computation reuse in our production workloads.

2.3 CloudViews Overview

Figure 4 illustrates an example scenario of computation reuse across three different users who are analyzing the same shared datasets (which include Customer, Sales, and Parts tables) and analyzing the sales behavior in the same Asia region. We can see how the insights that they are looking for is expressed as SQL queries that get compiled into query plans, and even though the user queries might appear very different, their query plans turn out to have significant portions in common (shown in orange and blue boxes). The analysts may identify that they are all analyzing the same market segment, i.e., the plain sentence version of the insights they are looking for has "In Asia" in common. This may lead them to create indexes or vertical partitions on the customer table. However, the query plans in the bottom of Figure 4a

[1]A virtual cluster represents a sub-cluster that is dedicated for one particular customer or business unit.

shows much large computations, called *query subexpressions*, are shared across them. Furthermore, different sets of computations are shared across different sets of users. To really discover these opportunities, the analysts would have to understand their analytics more deeply and then coordinate amongst themselves to manually share the common computations, requiring a lot of manual effort that is simply not possible at enterprise scale.

CloudViews addresses the above challenges by introducing automatic computation reuse in an online fashion. Figure 5 shows the overall system architecture of CloudViews and the various steps involved. We briefly summarize them below. CloudViews leverages the presence of large workloads in modern clouds [25]. It extracts the query workload into a denormalized subexpressions table that pre-joins the logical query subexpressions with their runtime metrics as seen in the history. Thereafter, we identify the common subexpressions across queries using a strict subexpression hash, known as *signature*, that uniquely captures a subexpression instance including its inputs used. From the set of all common subexpressions, we select the useful set of expressions to materialize and reuse. Some of the considerations for reuse include storage cost for materialization, processing time saved when reused, saving opportunities per customer, and the presence of concurrent queries that may not benefit from materialization-based reuse. Users can provide storage and other constraints (e.g., maximum number of views to create) for view selection. The view selection output is also made available to customers for insights and expected overall benefits. For the selected views, we collect their corresponding *recurring signatures* that discard time varying attributes like parameter values and input GUIDs, and are likely to remain the same in future instances of the recurring workloads. For easier lookup and control, we generate tags for each of the signatures that help fetch relevant signatures for a given SCOPE job and could also be used for access control. These tagged signatures are then polled by insights service and stored using Azure SQL databases. We also generate a query annotations file with the selected signatures that could be used for quickly debugging any job. For instance, in case of a customer incident, we can reproduce the compute reuse behavior by compiling a job with the annotations file.

At query time, for an incoming SCOPE job, the compiler extracts its tags and fetches the annotations from the insights service. These annotations are then parsed and stored in the optimizer context. During core search, the optimizer tries to match top down (match larger subexpressions first) whether any of the query subexpressions is already materialized. If yes, then it modifies the query plan to reuse the common subexpression with scan over previously materialized subexpression, updates more accurate statistics, and inserts the modified plan into the memo for overall costing. The plan using a materialized subexpression is chosen only if its cost is lower than the plan without the materialized subexpression. In either case, there is a follow-up optimization phase to check (in bottom-up manner) if any of the subexpressions are candidates for materialization. If yes, then an exclusive lock is obtained from the insights service and a spool operator with two consumers is added to that subexpression: one feeds into the rest of the query processing while the other materializes the common subexpression to stable storage. During execution, the job manager makes the view available even before the query finishes (referred to as early sealing in Cosmos), and notifies the insight service to release the view creation lock and start reusing it wherever possible. The modified query plans are surfaced to the users in the query monitoring tool and also logged into the telemetry for future analyses.

We refer interested readers to [26] for more details on each of the components in the above architecture.

2.4 Design Decisions and Limitations

We now highlight some of the key decisions that have served our workloads well while discussing some of the key limitations of our approach. The key things that worked well include:

- **Preserving query boundaries.** One way to reuse computation could be to combine query plans of overlapping SCOPE jobs into merged query plans. However, this is inconvenient since it requires changes to submission system and hard to explain the cost of merge query to different users. Also, fault tolerance becomes more intertwined due to the hard dependency between multiple jobs. CloudViews keeps job boundaries intact while opportunistically reusing computations wherever possible.
- **Online materialization.** CloudViews materializes common computations in an online manner as part of query processing, i.e., it does not require any offline cycles for materialization. As a result, CloudViews is easier to manage operationally without requiring uses to submit additional queries while also hiding the materialization latencies in large complex query DAGs.
- **Just-in-time views.** CloudViews are also materialized just-in-time when the first query hits a particular query subexpression instance. This means that: (i) the storage space for materialized views is consumed only when the views are about to be reused, and (ii) if the workload changes and a selected subexpression is no longer found in the workload then it will automatically stop being materialized.
- **Accurate cost estimates.** Traditional view selection approaches suffer from poor cardinality and cost estimates in query optimizers [40, 43] since they explore alternate query plan expressions that may not have been executed in the past. However, by considering only the same logical subexpressions for reuse, CloudViews is able to leverage the actual runtime statistics seen in the past instances of those subexpressions. As a result, it can make better decisions about the views that could improve performance when reused and the storage costs associated with them.
- **Scalable view selection.** View selection over large workloads is non-trivial since it explodes the search space exponentially. This is because traditional view selection algorithms consider more generalized sets of views, ones that may not have appeared in past query executions, and select the most useful ones from them. However, by restricting to common subexpressions, CloudViews can run subexpressions selection to Cosmos scale by running it as a label propagation problem in a distributed manner [24].
- **Lightweight view matching.** Traditional view matching require expensive containment checks during query optimization to determine whether a query could be answered from a view or not. CloudViews replaces that with lightweight hash equality checks that only require to recursively compute a signature for each subexpression and then match them with the signatures of one or more available views.

Some of the limitations of our approach are as follows:

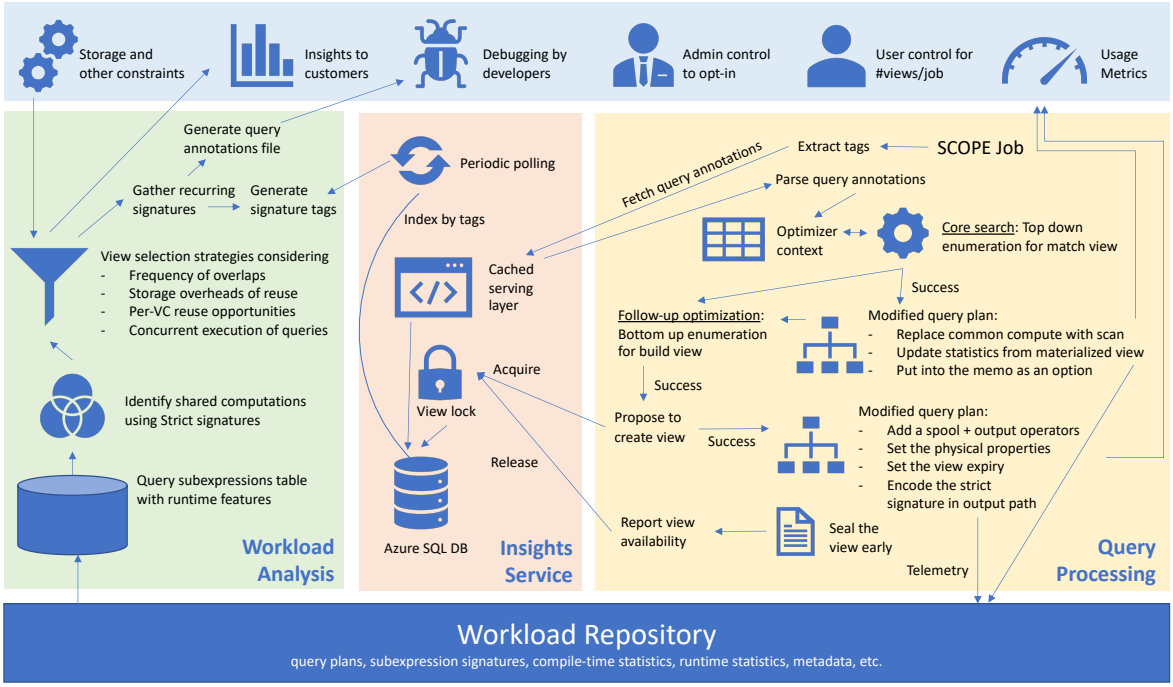


Figure 5: The CloudViews architecture for computation reuse in Cosmos.

- **Exact logical subexpression match.** The most obvious limitation of CloudViews is that it can only reuse the exact same logical query subexpressions, although they can have different physical implementations. While there are numerous opportunities for such reuse, there is even more potential for creating more generalized views and reusing them in queries that are contained in those views. We discuss this further in Section 5.
- **Concurrent queries.** CloudViews requires materializing common subexpressions before they can be reused in subsequent queries. Although the materialized views are made available as soon as the subexpression portion of the first query finishes (early sealing), still CloudViews cannot help queries that are submitted concurrently unless their submission schedule is altered, which is typically harder in production environments.
- **Not maintained.** CloudViews are treated as cheap throw-away views that are recreated whenever the inputs change. While this eradicates the need for view maintenance strategies, it also results in recreating views over sets of inputs where most have remained unchanged. This is particularly true for recurring queries with a sliding window, e.g., last seven days, where all except the most recent input in the window might remain same.
- **No DDL for user visibility or tuning.** CloudViews are created transparently to the users without registering them as any DDL statements. As a result, users do not have direct visibility into the catalog of available views at any given point nor can they reason about them. They can, however, see the CloudViews-generated files, given that they are stored in user-specified location, and even purge views whenever necessary.
- **User expectations.** CloudViews causes the first query hitting a common subexpression to slow down due to additional materialization overhead. Therefore, the users need to be informed about some queries getting impacted for overall workload efficiency.

Jobs	257,068
Pipelines	619
Virtual Clusters	21
Runtime Versions	12
Views Created	58,060
Views Used	344,966
Latency Improvement	33.97%
Processing Time Improvement	38.96%
Bonus Processing Time Improvement	45.01%
Containers Count Improvement	35.76%
Input Size Improvement	36.38%
Data Read Improvement	38.84%
Queuing Length Improvement	12.87%

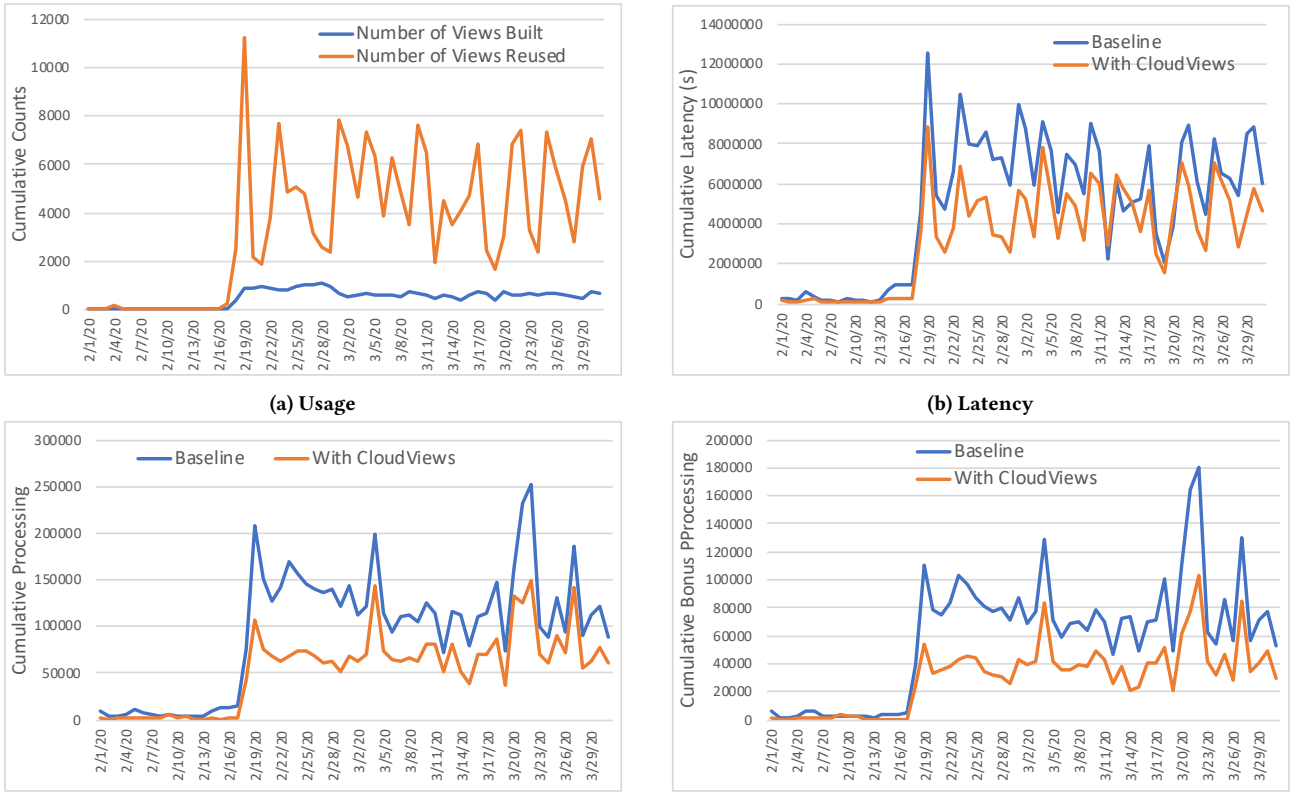
Table 1: Production Impact Summary

3 PRODUCTION IMPACT

In this section, we describe the impact of CloudViews when deployed for several customers over a two-month window (February–March 2020). Table 1 shows the summary of workload and the performance impact. Below we discuss them in more detail.

3.1 Usage

Let us first look at the usage numbers. Our current deployment strategy was *opt-in*, i.e., the feature was made available for customers and they can choose to enable it on their virtual clusters. Overall, we see more than 250k analytical SCOPE jobs across 21 virtual clusters using CloudViews. These jobs were from 619 unique data pipelines and ran over 12 different SCOPE runtime versions. Figure 6a shows the number of views materialized and



(a) Usage

(b) Latency

(c) Processing Time

(d) Bonus Processing Time

Figure 6: The usage and impact of CloudViews on production workloads.

reused over a two-month window. After an initial customer onboarding period, we see a periodic view creation and reuse pattern. Naturally, much more views are reused than created every day. Overall, approximately 58k views were created and they were reused 350k times, each view being reused almost 6 times on average. Our current eviction policies expire each of the views after one week of creation, thus consuming a fixed amount of storage (that is configured by the customers and affects the number of views selected for reuse) in the stable state.

3.2 Latency

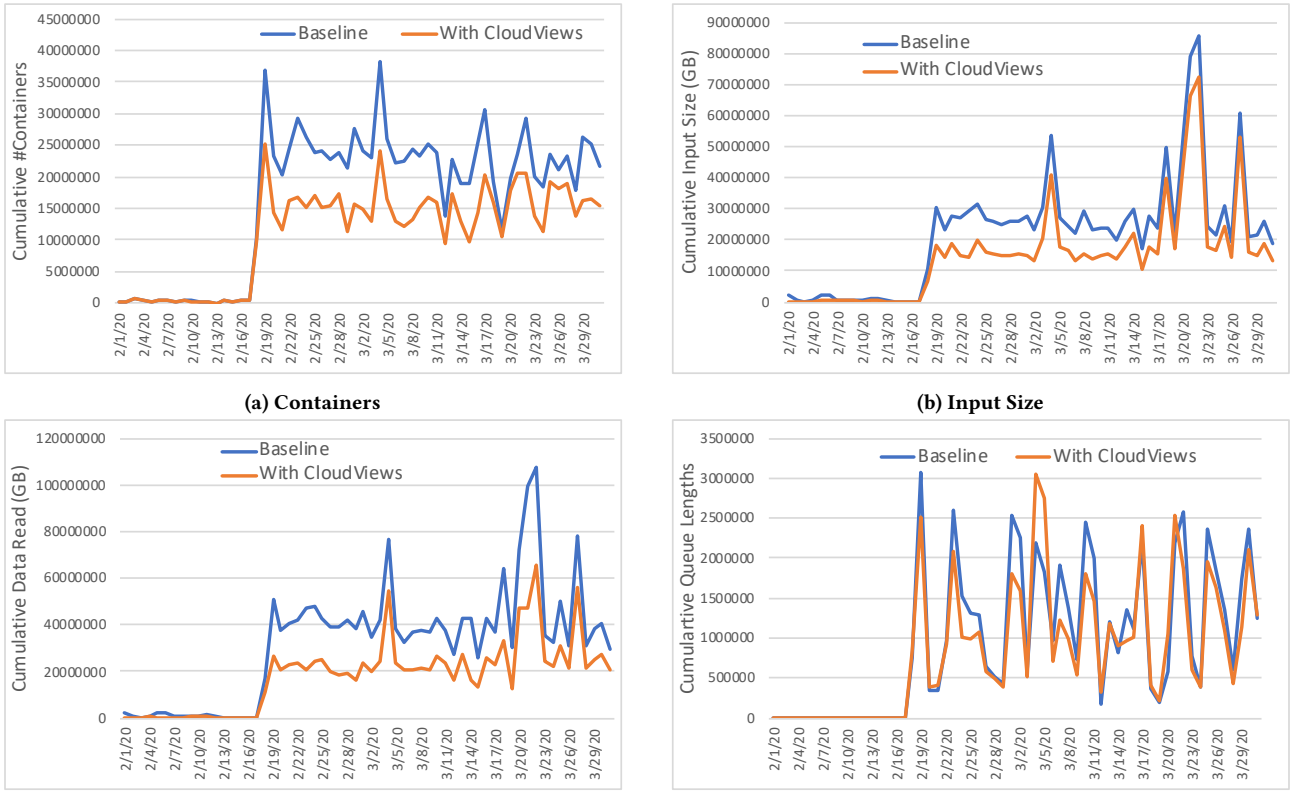
Latency is a crucial metric for customers and so we next explore the impact of CloudViews on job latencies. Figure 6b shows the daily cumulative latencies of jobs when CloudViews was enabled, compared to if it was not enabled, over the same two-month window. We can see that even though view materialization incurs an overhead, overall there is a gain in cumulative latencies of the jobs. This is because we materialize CloudViews in an online fashion in a separate stage that runs in parallel and hence the impact of latency is typically less. At the same time, we also see that the latency improvements are staggered and minimal on several days. This is because computation reuse could only improve latency if the portions of the query graph that was reused lies on the critical path of the job. Given that our view selection strategies do not optimize for that (since the objective function is to maximize for total compute), latency improvements are not guaranteed, especially with large query DAGs where overlapping computations may not be on the critical path. Still, we see a median per-job latency improvement of 15% and a cumulative overall improvement of close to 34%, that is significant for improving the customer experience in production workloads.

3.3 Processing Time

We now look at the impact on total processing time (i.e., the sum of processing time of all the containers used in the jobs) which is often considered a better measure of compute efficiency. Figure 6c shows the cumulative processing costs per day when CloudViews was enabled compared to if it was not enabled. Indeed, in contrast to latency, we can see more distinct change in processing time, with close to 39% improvement overall. This is because processing time savings do not depend on the critical path and any reuse in the query graph contributes to some savings, modulo the time spent in reading the materialized shared computation. Strictly speaking we also need to discount the extra processing time spent in writing the shared computation in the first job. Since we look at the observed processing times in the cluster all of these overheads automatically get accounted for. The processing time savings validate the utility of CloudViews to improve cluster efficiency and to free up spare resources that will let developers do more with the same set of resources.

3.4 Bonus Processing Time

Cosmos employs an opportunistic resource allocation policy to improve cluster utilization [8]. The idea is to allocate unused resources opportunistically to jobs in case they could use them, e.g., one or more stages in a job has more partitions than the number of containers available to the job or stages that could start making



(a) Containers **(b) Input Size**

(c) Data Read **(d) Queuing Lengths**

Figure 7: Other non-obvious impact of CloudViews on production workloads.

progress in parallel. We record the processing time using opportunistic resources as bonus processing time, and this is helpful to not only improve the utilization but to also improve SCOPE job performance by dynamically leveraging unused cluster capacity. Unfortunately, bonus processing also leads to unpredictability in performance, i.e., the job runtimes may vary a lot based on the unused capacity at any given time in the cluster. Therefore, reducing the reliance on bonus processing is desirable for more predictable job behavior. Interestingly, it turns out that by sharing common computations and not re-executing them with a lot of variance each time, CloudViews can reduce the reliance on bonus processing and hence improve job predictability. Figure 6d shows the reduction in bonus processing time with CloudViews. We can see a significant change in bonus processing time with an overall reduction of 45%, which is important for improving the overall system reliability.

3.5 Containers

While latency and processing are expected to be impacted by computation reuse in CloudViews, we also discover other rather unexpected implications. In particular, the total number of containers used in each SCOPE job is an important measure of resource consumption and computation reuse can also help improve that. Figure 7a shows the change in the cumulative number of containers used with CloudViews. We can see that similar to the total processing time, total number of containers also show significant improvement both for each day (and also for each job) and also overall (36% fewer containers used). This is because eliminating re-computation of large expensive chunks of query DAGs also eliminates the corresponding set of resources used to process those computations. Furthermore, due to the challenges in estimating cardinality over big data workloads, SCOPE query engine often ends up overestimating cardinalities and thus over-partitioning the intermediate outputs, leading to many more containers getting instantiated and each processing relatively smaller amounts of data [43]. Computation reuse automatically circumvents this issue by avoiding re-execution of such common computations in the first place and thus saving resources that would been otherwise be consumed due to mis-estimation. In fact, computation reuse further helps feed more accurate statistics from the previously materialized subexpressions to the rest of the query plan.

3.6 Input

Another less anticipated effect of computation reuse is the size of the input read. Figure 7b shows a sizeable reduction in the total inputs sizes read by all queries in that workload. This is because quite often the inputs datasets are filtered, selectively joined, or aggregated before they are materialized as common subexpressions, which end up being much smaller than the initial input sizes. Smaller input sizes not only reduce IO but also improve the container efficiency since SCOPE jobs are widest at the beginning (due to very large input sizes) and then their width drops significantly, causing lots of allocated containers remaining unused in a large portion of the query [7]. Smaller input sizes avoid such extreme container usage over the course of a job execution. Furthermore, it also eases the pressure on the storage layer, which is increasingly disaggregated from the compute in modern clouds [35], and thereby helps in reducing the throttling during peak load conditions.

3.7 Data Read

Apart from the input data read, let us also look at the total data read. Figure 7c shows the change in the overall data read and it is very similar to the trend of input read, although overall, data read improves by 39%, which is more than the improvements in input read. Reducing data reads eases the overall IO pressure including both the persistent store and also the local temporary store that is used to write intermediate outputs for each of the SCOPE jobs. This is significant because there could be hotspots with significant intermediate IO for large SCOPE DAGs and fewer data read can alleviate some of these hotspots.

3.8 Queue Lengths

SCOPE jobs are processed in a job service form factor, where users submit their jobs and they are queued until there are enough resources available for them to be scheduled. Interestingly, computation reuse can even help reduce the queue length due to less computations being done by each job which causes them to finish faster. Figure 7d shows the cumulative queue lengths seen by all jobs for each day in the workload, with a reduction in queue lengths on several days and an overall reduction of 13%. Shorter queue lengths improve the user experience and reduces their time to insights, enabling tighter SLAs in many cases. It also helps execute queries with more recently optimized query plans, e.g., leveraging more recently generated CloudViews, that would otherwise be missed by jobs sitting in longer queues.

4 OPERATIONAL CHALLENGES

In this section, we describe some of the operational challenges that we have faced. Note that these exclude the multiple rounds of customer feedback and feature improvements we did before deploying CloudViews in production.

- **Schedule-aware views.** Cosmos customers have built several workflow tools to schedule and monitor SCOPE jobs periodically. In some cases, these tools trigger all jobs at the start of every period, with the goal of finishing them before the period ends and to avoid missing their SLAs. Given that CloudViews requires materializing the common computation before it can be reused, jobs that get scheduled (and thus compiled) at the same time cannot benefit from such reuse. One option could be to alter the job submission and add a little lag to jobs that could reuse common computations. However, it turned out to be very hard to convince customers to change their job submission schedules. Instead, we modified our view selection algorithms to account for concurrent job submissions; specifically, we only consider subexpressions that could finish materializing before the start of other consuming jobs.
- **Per-customer view selection.** The data cooking process could create shared datasets across multiple customer virtual clusters (VCs), leading to computation reuse opportunities across multiple VCs as well. However, customers often care about the reuse and performance improvements in each of their individual VCs. This is because they want to benefit from better SLAs and do more processing on a per-VC basis. Furthermore, the cost of storing the common computations could be significant (depending on how much reuse is done) and customers often want to keep it separate for each VC. This means that selecting the views to materialize globally is not enough but rather we need to select them for each virtual cluster. Given that there are thousands of virtual clusters in Cosmos, it is not possible to run view selection separately for each of them. At the same time, a single view selection script that partitions workloads by virtual clusters needs to also consider the constraints per VC (see top left in Figure 5). It also makes it hard to maintain the single script with evolving requirements for different customers.
- **Signature correctness.** The core of CloudViews relies on a signature (i.e., a hash) to identify logical subexpressions. While this is trivial for native operators in the SCOPE engine, things can get murky with the user defined operators (UDOs) and the libraries used in it. In particular, it could be very difficult to compute the signatures in user code that involves recursively dependent libraries (there are extreme cases in Cosmos with very deep dependency chains). Traversing these long chains could slow down the entire compilation process. Furthermore, in some extreme cases, the UDOs may even contain non-determinism by design, e.g., `DateTime.Now`, `UTCNow`, `Guid.NewGuid()`, or `new Random().Next()`. It is not clear what the correct semantics are when computing signatures over such UDOs. Our approach is to exclude such extreme cases to avoid failures or incorrect results, i.e., we skip any computation reuse if the dependency chain is too long or if a UDO is found to contain non-determinism.
- **Impact of changed signatures.** While it is understood that the signatures will change with the workloads, sometimes they also evolve with new SCOPE runtime (due to changes in compilation, optimizer representation, or other code changes). As a result, all existing materialized views get invalidated. Thus, evolving signatures is very tricky since we need to keep track of changes that can affect signatures and re-run any prior workload analysis.
- **Other dependencies.** From Figure 5, we can see that CloudViews depends on other components, such as the offline workload analysis, insights service, the job manager to seal the view early, and the various customer interactions. However, these components often evolve independently and so they need to be kept in sync. For instance, the early sealing had to be ported to new job manager versions, the view storage locations need to be migrated to ADLS [13], and the insights service had to be scaled.
- **Handling GDPR requirements.** The emergence of new privacy regulations, such as [16], mean that we cannot simply look at the input paths but need to also keep track of when the inputs have changed due to *forget* requests and automatically stop consuming them henceforth. When inputs change, Cosmos handles updates as incremental files or delta updates. We handled input changes by ensuring that the input GUIDs are updated both with recurring updates and with GDPR related updates, which are handled separately in our storage layer.
- **Opt-in vs opt-out.** Given that CloudViews trades computation reuse for storage costs (and some latency overheads in the first job that hits the common computation), we need to make customers aware of the expected costs and benefits. As a result, we adopted an *opt-in* model of deployment where only the customers who bought in were onboarded. This also helped us address bug fixes and other deployment issues more gradually. However, *opt-in* also requires significant customer interaction eating up a lot

of program manager time. As a result, it is not scalable to large number of customers. Therefore, after sufficient hardening of the CloudViews feature in production, we have now started enabling it using an *opt-out* model, where virtual clusters are grouped into tiers (based on business importance) and they are automatically onboarded tier by tier, starting with the lowest tier.

- **Multi-level control.** We ended up placing several levels of control to enable or disable CloudViews. These include job-level control for individual developers to toggle CloudViews in their jobs, VC-level control for enabling or disabling specific VCs once they decide to onboard or opt-out, cluster-level to make the feature enabled or disabled across the board for the entire cluster instead of doing it for each of the thousands of VCs on each cluster, and insight service level control as the uber control for gate keeping and toggling during customer incidents.
- **Measuring impact.** Finally, while it is easy to measure performance improvements in a pre-production environment by re-running both the baseline and the modified version, it is less simple to measure performance once a feature is deployed in production. This is because it is very difficult to draw the baseline in a constantly changing production environment, e.g., when input sizes in recurring jobs change significantly [40]. It is also not possible to re-run all production jobs with CloudViews disabled to establish a baseline. Therefore, we took the following approach to identify baseline performance: we took previous instances of the queries that qualified for CloudView optimization and collected four weeks' worth of observations before enabling CloudViews on them. Thereafter, we took the 75^{th} percentile value of each of the performance metrics, such as latency and processing time, and compared them with each of the newer instance of that query once CloudViews was enabled.

5 LOOKING BACK AND FORTH

We now reflect back on our journey from research to production and discuss many of the next set of problems we see in the area of computation reuse.

5.1 From Research to Production

CloudViews has come a long way from research to production, starting from our initial research ideas in 2015 and then us spending the next five years for prototyping in the SCOPE codebase, rallying product team support around it, getting valuable customer feedback, fixing many of the bugs that were discovered, onboarding customers first by opt-in and later by opt-out approach, and finally addressing many of the operational challenges. More significantly, however, CloudViews introduced a self-tuning feedback loop to the SCOPE query engine, a significant departure from using just the compile-time estimates to leveraging how things went in the past for query optimization. Along the way, we also learned valuable lessons for doing applied research within a product group setting [23]. A key amongst them is the realization that productization of research ideas is often a much longer and sometimes even a painful journey, that requires perseverance and willingness to adapt. This is because production features need to consider a lot of corner cases. The new DevOps model where there is no dedicated testing team anymore, combined with the cloud service form factor of modern software, acts as a forcing function for product teams to put all requisite safeguards for a consistent user experience and lower maintenance overhead. Finally, applied research turns out to be a highly collaborative endeavor, right in the trenches of product teams, and therefore it needs to be appreciated as such.

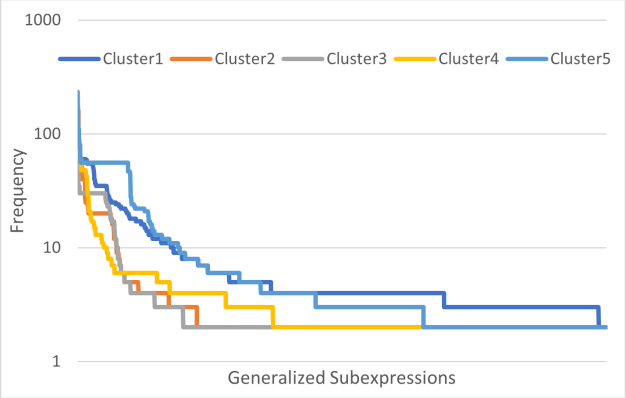


Figure 8: Opportunities for more generalized views: the x-axis shows the subexpressions that join the same sets of inputs, and the y-axis shows their corresponding frequency.

5.2 Towards Broader Workload Optimization

CloudViews helped open up the area of workload optimization for cloud query engines, leveraging the large workload telemetry that are visible, with appropriate anonymizations, in modern cloud environments [25]. This resulted in a mindset change from optimizing just a query at a time to also consider optimizing the entire workload, something which customers care a lot in order to manage their total cost of ownership (TCO). Specifically, the notion of signatures to uniquely identify query subexpressions turned out to be very helpful not just for computation reuse, but also for applications such as discovering interesting query patterns in the workload, learning high accuracy micro-models for specific portions of the workload [27], compressing workloads into a representative set for pre-production evaluation, and surfacing data and job dependencies for interesting pipeline optimizations. Likewise, the insights service evolved into an independent component that could serve many different kinds of insights, e.g., cardinality [43], cost [40], or resources [39]. The insights could be scaled and bulk loaded upfront to the SCOPE optimizer, with an end to round trip latency of around 15 milliseconds. All of the above resulted in common pieces of infrastructure that could be leveraged across several new optimizations in SCOPE, and even for other query engines like Spark.

5.3 Generalized Reuse

CloudViews identifies view candidates using per-operator signature matching, which establishes syntactic equivalence but does not consider views that differ syntactically but are otherwise logically equivalent (e.g., `SELECT * FROM Sales WHERE CustomerId > 5` and `SELECT * FROM Sales WHERE 2 * CustomerId > 10`). However, while relaxing this constraint may offer additional performance improvements by enabling the reuse of more views, computing logical equivalence is expensive and in general undecidable. Nonetheless, recent work has pushed down this cost to the point where exploiting these opportunities may be feasible for many

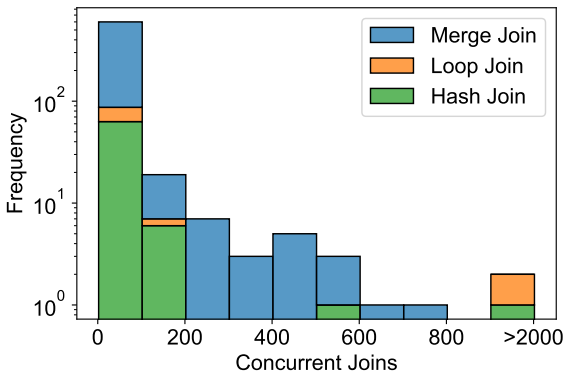


Figure 9: Concurrently executing joins on a Cosmos cluster occurring in a single day. This workload contained two outliers that were concurrently executed 2016 and 23,040 times.

queries [10, 11, 38, 48, 49]. Evaluating this precise trade-off, and quantifying the number of materialized views that more general logical equivalence would allow, remains a ripe avenue for future research.

Similarly, equivalence itself (logical or otherwise) is more restrictive than necessary for view reuse, and in many cases containment may offer similar opportunities for reusing computation (e.g., materializing `SELECT * FROM Sales WHERE CustomerId > 5` and using it to answer the query `SELECT * FROM Sales WHERE CustomerId > 6`). Like logical equivalence, containment is a hard problem (in general NP-complete, although polynomial-time algorithms exist for some subproblems), has been extensively explored in the literature, and is expensive to compute. Nonetheless, some recent work has used machine learning to evaluate containment rates [22], and many techniques used to efficiently compute logical equivalence may be applicable to containment as well.

Figure 8 shows the reuse opportunity when considering one kind of generalization: subexpressions that join the same sets of inputs. These subexpressions could still have different projections, selections, or group by operations, which could be merged to create more general materialized views and then later query could be rewritten using containment checks. Figure 8 shows the opportunity over the same five clusters as in Figures 2 and 3, and we see lots of generalized subexpressions with frequencies on the order of 10s to 100s.

5.4 Reuse in Concurrent Queries

CloudViews materializes views for reuse in subsequent queries, which is necessary for queries that are temporally non-overlapping. However, opportunities for reuse exist for *concurrent* queries, which does not require pre-materialization since intermediate results may be directly pipelined. While at first this might seem like an infrequent occurrence, we observed thousands of such opportunities per day in our production workloads. Figure 9 illustrates these opportunities for concurrently-executing joins within a single Cosmos cluster over a single day. We see that several join instances that are found to be concurrent hundreds to thousands of times.

Extending CloudViews to support concurrently executing queries and extending its feedback loop to efficiently learn the trade-offs between immediate reuse and materialization remains a ripe direction for future exploration.

5.5 Reuse in Other Engines

The idea of computation reuse goes beyond the SCOPE query engine. In fact, we adapted the computation reuse ideas to the Spark query engine as part of the SparkCruise project [36]. SparkCruise selects high utility common computations and performs automatic materialization and reuse for Spark SQL queries. Like CloudViews, SparkCruise analyzes past application workload logs to select common subexpressions for reuse. The list of common subexpressions is provided to the Spark query optimizer for future materialization and reuse. All these actions are performed automatically without any active involvement from the customer. On TPC-DS benchmarks, SparkCruise can reduce the running time by approximately 30% [36].

Even though both CloudViews and SparkCruise share the same ideas, there are differences in the target systems and the deployment environments. SCOPE query engine is developed by Microsoft and we added the signatures and optimizer rules deep inside the query optimizer. However, Spark is an open-source project and making any code changes in Spark will tie us to a specific version and delay the upgrade process in the future. Therefore, we use the optimizer extensions API in Spark to add two additional rules to the query optimizer — first for online materialization, and second for computation reuse. We also implemented an event listener for Spark SQL that can log query plans and compute signature annotations on the logical query plan object. The user simply needs to add SparkCruise library and set a couple of configuration parameters. With these changes, we can provide computation reuse in Spark without modifying its code.

Currently, SparkCruise is deployed on Azure HDInsight [32]. Azure HDInsight offers Spark cluster-as-a-service. Users can spin up ephemeral Spark clusters, run their query workloads, and delete the cluster after the job has finished. This scenario requires a fast workload-based feedback loop. To enable this fast feedback loop, we gave the control of the workflow to the end users or the data engineers. The users can schedule the workload analysis and view selection job periodically, or as often as the changes in the query workload. To help users understand their query workloads and decide whether SparkCruise will benefit their workload or not, we provide an interactive Workload Insights Notebook in Python [31]. The Workload Insight Notebook shows the workload statistics in aggregate as well as the redundancies in the workload. The results from the notebook can convince the users to enable the computation reuse feature on their workloads.

CloudViews and SparkCruise show that the benefits of computation reuse are not limited to a specific query engine. We believe that computation reuse should be a fundamental principle, like data locality and fault tolerance, when designing big data processing systems.

5.6 Other Applications of Reuse

The CloudViews mechanism of producing artifacts as part of query execution is useful in many other related applications:

- **Checkpointing.** Computation reuse can be applied for automatic checkpoint and restart in large analytical queries. The idea is to select intermediate subexpressions in a job's query plan to materialize and reuse them in case the job is

restarted after a failure. Job failures are common in production clusters, with a small fraction failing every day. There are many reasons for failures including but not limited to storage errors, lack of compute capacity, missing input files, and network timeouts [46]. These transient errors are especially problematic for long running jobs that run for hours and fail towards the end. Typically, the failed jobs are resubmitted after a short delay. However, these jobs execute from the start all over again, thus wasting valuable compute resources and also delaying the final results. CloudViews can be used in this scenario to recover quickly from failures. During the compilation phase, we use query history to find which operators are more likely to fail and add a checkpoint just before them [50]. Then, during the resubmission, CloudViews can load the last available checkpoint thereby avoiding re-computation. Going forward, it would be interesting to select views that increase cluster utilization by maximizing the reuse across queries and minimizing the recovery time at the same time.

- **Pipeline Optimization.** Enterprise data analytics consists of data pipelines where analytical queries are interconnected by their outputs and inputs. Furthermore, the output of each producer query in the pipeline is typically consumed by multiple downstream queries. Unfortunately, the producers are not aware of the right data representations, or physical designs, required by their consumers. As a result, downstream queries need to prepare the data before they can run the actual processing. Therefore, it is crucial to leverage the data dependencies for creating the right physical designs tailored to the downstream queries. This can be done by producing the right physical design as part of query execution of producer job, i.e., a CloudViews that captures the required physical designs needed by downstream jobs.
- **Sampling.** Sampling is a powerful technique used for approximate query execution. Approximate query execution helps lower the latency and cost of running complex analytical queries on large datasets [29]. CloudViews style computation reuse can be applied for reducing the cost of approximate query execution even further. This can be achieved by sampling the views created by CloudViews. Sampled views will particularly help reduce query latency and cost in queries where substantial work happens after the sampler. Likewise, we could create statistics on the common subexpressions to provide insights to data scientists and analysts.
- **Bit-vector Filtering.** Bit-vector filters such as bitmap filters, Bloom filters and similar variants have been proposed by both industry and academia to perform semi-join reductions [15]. Semi-join reductions help filter rows which do not qualify join condition early-on in the query execution plan. Bit-vector filters have a low storage and compute overhead. They are commonly used in hash joins during query processing. CloudViews style computation reuse can be applied for generating bit-vectors during query execution as well. During query execution, a spool operator could be used for generating the bit-vector filter from right child of hash join and reuse it in subsequent queries.

6 OTHER RELATED WORK

Compute reuse is a hot topic in industry and we discuss some of the trends in other major companies below.

Snowflake is a cloud-based data warehouse company. The Snowflake design caches results of every query executed in the past 24 hours [41]. Users can then postprocess the cached result for further analysis. However, caching only the final results has limited applicability. CloudViews on the other hand reuses computation at any point in the query plan and is thus a superset of result set caching performed by Snowflake.

Google BigQuery is an interactive big data system that supports different kinds of caching and computation reuse. In particular, similar to Snowflake, it also supports caching arbitrary query results [18]. However, same as Snowflake, users have to manually rewrite their queries against the cached query results. BigQuery also supports materialized views that are maintained and support automatic query rewriting [17]. However, it only supports queries with aggregate function to simply the query rewriting problem.

Amazon Redshift is cloud data warehouse product from Amazon. Redshift enables materialized views to be automatically refreshed with automatic query rewrites [3]. However, the materialized views are still created manually and unlike online materialization in CloudViews, the responsibility of creating materialized views in Redshift lies with the user.

Alibaba [2] allows users to create materialized views. Once created, the materialized views could be used in queries. However, Alibaba's data warehouse engine requires manual re-writes to reuse materialized views. Again, unlike the automatic materialization and reuse in CloudViews.

Oracle deploys an algorithm called extended covering subexpressions (ECSE) to select materialized views for reuse [1]. It performs pairwise selection of 'join sets' and deploys other heuristics to reduce the search space to polynomial size. The authors test ECSE against a two-node configuration against a small number of queries, and intend to deploy the technique to their cloud in the future. By contrast, CloudViews is not restricted to this smaller class of candidate materialized views and has been continuously executing at extreme scale over hundreds of thousands of queries per day.

Finally, there are several other works that apply computation reuse in other settings. Yuan et al. [45] apply computation reuse on query workloads from Alibaba. Inspired by BigSubs algorithm [24] of CloudViews, they formulate the subexpression selection problem as ILP (Integer Linear Programming) and use deep reinforcement learning to solve the ILP. AStream [30] is a shared computation reuse framework for streaming queries. AStream can dynamically adapt to changing streaming query workloads without affecting the query execution topology. Computation reuse was also applied on Machine Learning workloads by Helix [44]. Helix finds common intermediate computation between iterations and automatically materializes some of them for future iterations.

7 CONCLUSION

Large-scale data processing infrastructures are key to data-driven decisions in modern enterprises. Unfortunately, the scale and complexity of these infrastructures could also make them unwieldy and highly inefficient. In this paper, we describe how large scale data preparation, also referred to as data cooking, in the Cosmos big data infrastructure at Microsoft often leads to

redundant computations across different data pipelines and how computation reuse naturally augments the data cooking processing by fine-tuning the cooked datasets with more shareable ones. We show the impact of CloudViews, an automatic computation reuse infrastructure that we had build in the SCOPE query engine, over large production workloads and discuss many of the operational challenges that we faced. CloudViews has not only helped improve operational efficiency (37% less aggregated processing time) and customer experience (34% less aggregated latency), but it has also opened up new avenues and reusable infrastructure for a broad range of feedback-driven workload optimizations — the stepping stones towards a self-tuning intelligent cloud.

REFERENCES

[1] Rafi Ahmed, Randall G. Bello, Andrew Witkowski, and Praveen Kumar. 2020. Automated Generation of Materialized Views in Oracle. *VLDB* 13, 12 (2020), 3046–3058.

[2] alibaba [n.d.]. Create a materialized view. https://www.alibabacloud.com/help/doc-detail/116486.htm.

[3] amazon-redshift [n.d.]. Amazon Redshift announces automatic refresh and query rewrite for materialized views. https://aws.amazon.com/about-aws/whats-new/2020/11/amazon-redshift-announces-automatic-refresh-and-query-rewrite-for-materialized-views/.

[4] apache-hive [n.d.]. Apache Hive. https://hive.apache.org/.

[5] apache-spark [n.d.]. Apache Spark. https://spark.apache.org/.

[6] aws-athena [n.d.]. AWS Athena. https://aws.amazon.com/athena/.

[7] Malay Bag, Alekh Jindal, and Hiren Patel. 2020. Towards Plan-aware Resource Allocation in Serverless Query Processing. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[8] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *OSDI*. 285–300.

[9] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB* 1, 2 (2008), 1265–1276.

[10] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *VLDB* 11, 11 (2018), 1482–1495.

[11] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*.

[12] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. 2020. Unearthing inter-job dependencies for better cluster scheduling. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1205–1223.

[13] datalake [n.d.]. Azure Data Lake. https://azure.microsoft.com/en-us/solutions/data-lake/.

[14] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI'04)*. USENIX Association, USA, 10.

[15] Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. 2020. Bitvector-Aware Query Optimization for Decision Support Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. 2011–2026.

[16] gdpr-art17 [n.d.]. Art. 17 GDPR. Right to erasure ('right to be forgotten'). https://gdpr.eu/article-17-right-to-be-forgotten/.

[17] Google. 2020. *BigQuery Materialized View.* Retrieved November 18, 2020 from https://cloud.google.com/bigquery/docs/materialized-views-intro

[18] Google. 2020. *BigQuery Result Caching.* Retrieved November 18, 2020 from https://cloud.google.com/bigquery/docs/cached-results

[19] google-bigquery [n.d.]. Google BigQuery. https://cloud.google.com/bigquery.

[20] Himanshu Gupta and Inderpal Singh Mumick. 2005. Selection of Views to Materialize in a Data Warehouse. *IEEE Trans. Knowl. Data Eng.* 17, 1 (2005), 24–43.

[21] Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB J.* 10, 4 (2001), 270–294.

[22] Rojeh Hayek and Oded Shmueli. 2020. Improved Cardinality Estimation by Learning Queries Containment Rates. In *EDBT*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). 157–168.

[23] Alekh Jindal. 2020. Applied Research Lessons from CloudViews Project. *SIGMOD Record (to appear)* (2020).

[24] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Thou Shall Not Recompute: Selecting Subexpressions to Materialize at Datacenter Scale. In *VLDB*.

[25] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Jarod Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload Optimization for Cloud Query Engines. In *SoCC*.

[26] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *SIGMOD*.

[27] Alekh Jindal, Shi Qiao, Rathijit Sen, and Hiren Patel. 2020. Microlearner: A fine-grained Learning Optimizer for Big Data Workloads at Microsoft. *Under Submission* (2020).

[28] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 117–134.

[29] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex Ad-Hoc Queries in Big Data Clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2016)* (proceedings of the acm sigmod international conference on management of data (sigmod 2016) ed.). ACM - Association for Computing Machinery. https://www.microsoft.com/en-us/research/publication/quickr-lazily-approximating-complex-ad-hoc-queries-in-big-data-clusters/

[30] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AStream: Ad-Hoc Shared Stream Processing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 607–622. https://doi.org/10.1145/3299869.3319884

[31] Microsoft. 2020. *Azure SparkCruise Samples.* Retrieved November 18, 2020 from https://github.com/Azure-Samples/azure-sparkcruise-samples

[32] Microsoft. 2020. *SparkCruise on Azure HDInsight.* Retrieved November 18, 2020 from https://docs.microsoft.com/en-us/azure/hdinsight/spark/spark-cruise

[33] Nick. 2014. *Microsoft uses real time telemetry 'Asimov' to build, test and update Windows 9.* Retrieved November 13, 2020 from https://mywindowshub.com/microsoft-uses-real-time-telemetry-asimov-build-test-update-windows-9/

[34] Presto. 2020. *Presto.* Retrieved November 18, 2020 from https://prestodb.io/

[35] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. 2017. Azure Data Lake Store: a hyperscale distributed file service for Big Data analytics. In *SIGMOD*. 51–63.

[36] Abhishek Roy, Alekh Jindal, Hiren Patel, Ashit Gosalia, Subru Krishnan, and Carlo Curino. 2019. SparkCruise: Handsfree Computation Reuse in Spark. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1850–1853. https://doi.org/10.14778/3352063.3352082

[37] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*. 249–260.

[38] Guillem Rull, Philip A. Bernstein, Ivo Garcia dos Santos, Yannis Katsis, Sergey Melnik, and Ernest Teniente. 2013. Query containment in entity SQL. In *SIGMOD*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). 1169–1172.

[39] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. 2020. AutoToken: Predicting Peak Parallelism for Big Data Analytics at Microsoft. *PVLDB* 13, 12 (2020), 3326–3339. https://doi.org/10.14778/3415478.3415554

[40] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *SIGMOD*. 99–113. https://doi.org/10.1145/3318464.3380584

[41] Snowflake [n.d.]. Caching in Snowflake Data Warehouse. https://community.snowflake.com/s/article/Caching-in-Snowflake-Data-Warehouse.

[42] Z. Wang, K. Zeng, B. Huang, W. Chen, X. Cui, B. Wang, J. Liu, L. Fan, D. Qu, Z. Hou, and T. Guan. 2020. Tempura: a general cost-based optimizer framework for incremental data processing.. In *VLDB*. 14–27.

[43] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *PVLDB* 12, 3 (2018), 210–222.

[44] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. HELIX: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 446–460. https://doi.org/10.14778/3297753.3297763

[45] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1501–1512. https://doi.org/10.1109/ICDE48307.2020.00133

[46] Matei Zaharia. 2019. Lessons from Large-Scale Software as a Service at Databricks. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 101. https://doi.org/10.1145/3357223.3365870

[47] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *VLDB J.* 21, 5 (2012), 611–636.

[48] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. 2020. SPES: A Two-Stage Query Equivalence Verifier.

[49] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Dong Xu. 2019. Automated Verification of Query Equivalence Using Satisfiability Modulo Theories. *VLDB* 12, 11 (2019), 1276–1288.

[50] Yiwen Zhu, Alekh Jindal, Malay Bag, and Hiren Patel. 2020. Phoebe: A Data-Driven Checkpoint Optimizer. *Under Submission* (2020).

Industrial Paper

WILSON: A Divide and Conquer Approach for Fast and Effective News Timeline Summarization

Yiming Liao
The Pennsylvania State University
State College, PA, US
yiming@psu.edu

Shuguang Wang
The Washington Post
Washington, D.C., US
shuguang.wang@washpost.com

Dongwon Lee
The Pennsylvania State University
State College, PA, US
dongwon@psu.edu

ABSTRACT

Major news media frequently uses the method of *news timeline summarization* to summarize important daily news over major events across the timeline. While various sophisticated methods have been proposed to generate both concise and complete news timelines, in practice, generating timelines from a large number of news articles not only faces quality issues but also encounters the challenge of generation speed, which all existing methods have neglected. To mitigate these issues, in this work, we propose to speed up timeline generation by dividing the whole summarization task into sub-summarization tasks, adopting the "divide and conquer" philosophy: (1) date selection and (2) text summarization.

Furthermore, since existing methods in news timeline summarization pay less attention to the date selection than text summarization, in this paper, we re-examine the role of date selection in news timeline summarization and demonstrate that accurate date selection "alone" can significantly contribute to the task of news timeline summarization. Leveraging on the explicit date selection, then, we propose a simple yet fast and effective news timeline summarization method, named WILSON (neWs tImeLine SummarizatiON). Experimented on two widely used timeline summarization benchmark datasets, *timeline17* and *crisis*, empirical evaluation shows that WILSON outperforms state-of-the-art approaches in both speed and ROUGE scores, significantly improving ROUGE-2 F1 scores by 9.5%~17.7% and reducing generation time by two orders of magnitude. A further user study with professional journalists also validates the superiority of WILSON. Finally, we build a real-time news timeline summarization system and achieve encouraging results on an industrial-level corpus.

1 INTRODUCTION

Along with the rapid development of web services, an increasing number of news articles are published daily, describing both major and minor events worldwide. Due to the tremendous amount of news articles being produced every day, readers easily get lost in this information flood. Fortunately, *news timeline*, which summarizes each event with primary messages in a chronological order, makes it easy for readers to gain key insights and understand the evolution of news events. As such, many major news media has adopted the idea and have frequently produced news timelines of major news events. For example, Table 1 describes how 2018 North Korea-United States Singapore summit finally became a reality. Note that as the example illustrates, creating a news timeline requires the resolution of two sub-problems: (1) choosing of an ideal number of days among hundreds or thousands of candidate days, and (2) generating succinct text summaries per days.

Feb. 25, 2018 North Korea is "willing to have talks" with the United States, South Korea says, as the PyeongChang Winter Olympics close in a burst of fireworks and diplomacy.
Mar. 8, 2018 Trump agrees to meet Kim for talks, an extraordinary development after months of heightened tension. Kim commits to stopping nuclear and missile testing.
Mar. 27, 2018 Kim makes a clandestine visit to Beijing to discuss the negotiations with South Korea and the United States.
...
Jun. 1, 2018 Trump says the summit will take place June 12 as planned. During a visit to the White House, a North Korean hands Trump a large letter from Kim.

Table 1: An example timeline by The Washington Post about the summit between United States and North Korea.[1]

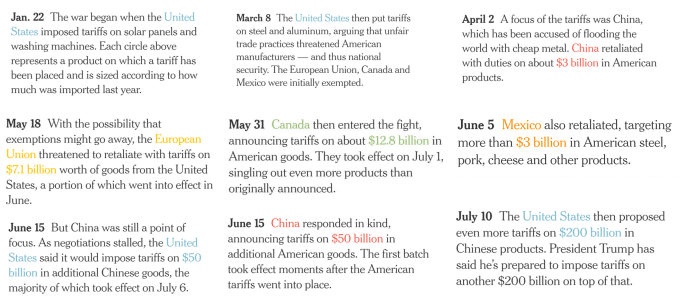
1.1 Industrial Use Case

Combined with visual or interactive interfaces, news timelines can provide a convenient way to compress overloaded news to audience. Figure 1 illustrates two real timeline example on two major US newspapers. Figure 1 (a) is an interactive timeline summarization about Trump-Russia investigation from The Washington Post, while Figure 1 (b) is a text-based timeline summarization about China-US Trade War from The New York Times. To help readers better understand the evolution of each news event, journalists take time to collect and organize related news articles, figure out major events and story lines, and "manually" summarize them in a chronological order. As events such as natural disasters and political issues can span from several months to multiple years and involve thousands of news articles, such a manual process cannot scale well. As this process is both time-consuming and labor-intensive, currently, despite the popularity of the concept, not all newspapers are able to quickly produce such news timelines.

To address this challenge, several automatic news timeline summarization methods have emerged in recent years [4, 12, 21, 22, 25, 27, 29]. By and large, there are mainly two categories of news timeline generation methods. One is aimed at separating

[1] https://www.washingtonpost.com/graphics/2018/national/trump-kim-jong-un-timeline/

(a) Trump-Russia investigation (The Washington Post)



(b) China-US Trade War (The New York Times)

Figure 1: News timeline summarization examples on major news media

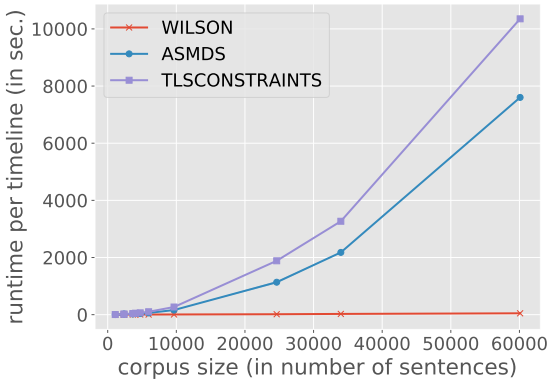


Figure 2: Running times over varying corpus sizes using *Timeline*17 and *Crisis*. ASMDS and TLSCONSTRAINTS are the two state-of-the-art methods that use submodular framework. Note that, as both of them are not sufficiently scalable to the large corpus, we followed [12] to use a reduced corpus here.

different stories from a whole news corpus, such as using variants of topic modeling [8, 31] and neural networks [30]. Another category focuses on generating a series of chronological summaries for one specific event from only relevant news articles [12, 22, 28], where the first categories can serve as pre-processing to find relevant news articles for each event. In this paper, our focus is on the latter category in an unsupervised manner.

However, majority of existing methods focus only on the quality of generated timelines and neglect the generation speed. For example, the state-of-the-art unsupervised approach adopts submodular framework [12] and requires the pairwise similarities for all tokenized sentences, which could be over 100,000 per timeline. This yields extremely slow running time, as clearly demonstrated in the comparison of running times in Figure 2. As the compression rates of timeline summarization vary with events and journalists may not know the exact value beforehand, iterative trials with different values are necessary, which makes faster timeline generation even more important. Therefore, in this work, we are greatly motivated by real industrial use cases to speed up the timeline generation by dividing the whole summarization tasks into multiple small summarization tasks by date separation.

News timelines are composed of both salient dates and daily summaries, but previous studies mainly focus on modeling relationships among article contexts while paying less attention to date selection. For example, some models [14, 24, 26, 27] just treat date information the same as text information and include it as one of the features, while others [4, 19] simply use date frequency to resolve events. Although simply modeling text correlation shows good performance on both timeline summarization and date selection [12], it is not clear how date selection will contribute to news timeline summarization. In addition, existing state-of-the-art unsupervised approaches mostly include global optimization, which helps daily summaries to be relevant to the topic. However, using global optimization also makes daily summaries less specific per each day and very time-intensive to generate timelines. Therefore, considering both the quality and speed of news timeline summarization, this paper makes the following main contributions:

(1) We re-examine the role of date selection in timeline summarization and show that, even without considering contextual correlation across different dates, accurate date selection is sufficient to generate high-quality news timelines. More importantly, although ignoring contextual correlation across dates leads to a lower empirical upper bound than other models, all of the previous approaches still fail to reach this lower upper bound, and they are not even close.
(2) Leveraging the explicit date selection, we propose a simple but fast and effective unsupervised news timeline summarization method, named WILSON. Experimented on two widely used timeline summarization datasets, WILSON outperforms state-of-the-art approaches in both speed and ROUGE scores, significantly improving ROUGE-2 F1 score by 9.5%~17.7% and reducing generation time by two orders of magnitude.
(3) To our best knowledge, WILSON is the first work to include an evaluation by professional journalists in news timeline summarization. Through manually comparing the machine-generated news timelines with corresponding human-generated ones, journalists confirm that our approach produces better timelines than competing methods.
(4) Based on the proposed WILSON, we build a real-time news timeline summarization system on an industrial-level news corpus.

2 THE PROPOSED METHOD: WILSON

In this section, we introduce our proposed method, named WILSON (neWs tImeLine SummarizatiON), also illustrated in Figure 3. Besides the pre-processing modules such as temporal tagging

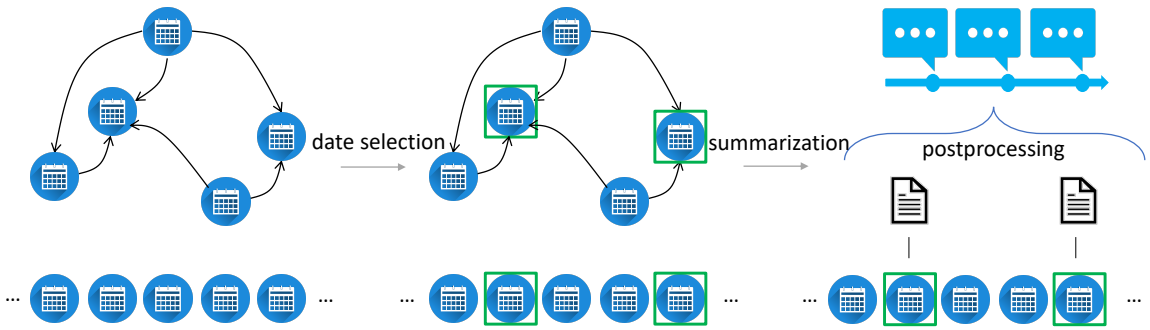


Figure 3: Workflow of our proposed method–WILSON.

and search engine indexing, WILSON mainly consists of two components – explicit date selection and text summarization for each selected date.

2.1 Problem Formulation

A news timeline can be viewed as a series of chronologically ordered daily summaries over main events, denoted by (d_i, S_i), where d_i and S_i stand for i_{th} date and i_{th} summary. Thus, news timeline summarization can be formulated as:

Definition 1 (News Timeline Summarization). *Given a corpus of articles C_q, which is associated with a topic query q and a time window $[t1, t2]$, the process of automatic timeline generation is to produce a series of daily summaries $(d_1, S_1), ..., (d_T, S_T)$, where $t1 \leq d_i \leq t2$.*

For both readability and reliability of generated news timelines, we follow existing works and utilize extractive summarization, which directly selects sentences from the corpus as summaries. More specifically,

Definition 2 (Extractive News Timeline Summarization). *Given a corpus of articles C_q, which is associated with a topic query q and a time window $[t1, t2]$, the corpus is first tokenized to dated sentences $\{(date_i, sentence_i) | date_i \in [t1, t2], sentence_i \in C_q\}$ by a date expression in the sentence and/or by the publication date, then the timeline generation is to produce a series of daily summaries $(d_1, S_1), ..., (d_T, S_T)$, where $t1 \leq d_i \leq t2$ and $S_i = (sentence_{i1}, ..., sentence_{iN})$.*

The number of selected dates T and sentences N are hyper-parameters and chosen by users to control the compression rate of the generated timelines. Date selection is evaluated by f1 scores and summaries are evaluated by ROUGE [10].

2.2 Date Selection

We use HeidelTime [20] to tag temporal expressions in sentences during pre-processing stage and start with an unsupervised date selection algorithm [23] to select the most salient dates: (1) we build a date reference graph with dates as nodes and reference relationships as edges; (2) then, we run the PageRank algorithm [16] on the graph and select the top T ranked nodes as the most salient dates. Date references refer to the sentences s_{ij} that are published on $date_i$ while mentioning $date_j$. We experiment with 4 types of edge weights as follows:

- W1: the number of reference sentences $|s_{ij}|$
- W2: temporal distance $|date_j - date_i|$ in days
- W3: W1 $*$ W2, which considers both frequency and temporal distance.
- W4: We adopt BM25 [18] to estimate the relevance of sentences to the query, and use max $BM25(s_{ij}, q)$ as edge weight for each reference.

Edge Weight	Date F1	Rouge-1 F1	Rouge-2 F1
timeline17			
W1	0.5512	0.3905	0.0969
W2	0.5528	0.4029	0.1002
W3	0.5628	0.4009	0.0995
W4	0.5068	0.3934	0.0934
crisis			
W1	0.3022	0.3476	0.0715
W2	0.2838	0.3604	0.0715
W3	0.2710	0.3575	0.0738
W4	0.2925	0.3509	0.0726

Table 2: Performance of different edge weights

For example, considering $date_i$=2018-06-01, $date_j$=2018-06-12, and s_{ij} composed of only two sentences, i.e. *Trump says summit with North Korea will take place on June 12* and *The summit will take place on June 12.* Then, W1 is the number of sentences and equals 2, while W2 is the difference between 2018-06-01 and 2018-06-12 in days and equals 11. Accordingly, W3 equals W1 $*$ W2 and is 22. For W4, we treat each sentence as a document, use topic query q to score each document with BM25, and take the maximum BM25 score as W4.

As Table 2 shows that all four edge weights yield comparable results, date reference relationship alone can extract as accurate date selections as topical information. Since constructing topical relationships across dates takes extra time, we finally adopt W3 as the edge weight to select the most salient dates in the rest of this paper. Note that, for completeness, we also generate daily summaries to obtain a complete news timeline per each date selection and evaluate the summaries by ROUGE scores in Table 2. The details about daily summarization is introduced in the next subsection.

Although the occurrence of an event signals its importance within the news timeline [4] and is well leveraged in existing timeline summarization algorithms, we note that the occurrence of events is also correlated with the recency of events, where past events occur earlier and are more heavily reported than recent events. Consequently, existing approaches may suffer from this issue. For example, approaches that optimize the summaries to

Date Selection	Date Coverage (±3)	Date F1	ROUGE-1	ROUGE-2	ROUGE-S*
Timeline17					
Uniform	0.8398	0.4475	0.3896	0.0917	0.1598
W3	0.7828	0.5668	0.4000	0.0995	0.1676
W3 + Recency	0.8111	0.5542	0.4036	0.1005	0.1702
Crisis					
Uniform	0.5932	0.1325	0.3387	0.0570	0.1138
W3	0.5459	0.2726	0.3573	0.0738	0.1246
W3 + Recency	0.5885	0.2748	0.3597	0.0760	0.1270

Table 3: Performance on date coverage

recover the whole corpus, such as ETS [29] and TILSE [12], will generate more summaries on past events.

In addition, as most references in articles refer to past events, the current date selection algorithm tends to give too much weight on old dates and will also result in timelines that lack recent dates. For a better illustration, we present the Cumulative Distribution Function (CDF) of the date duration between selected dates and the start date in Figure 4. As expected, both TILSE (Submodular) and date selection via PageRank (Tran et al.) tends to select more old dates, while the date distribution of ground-truth timelines is generally more uniform. Thus, we use the standard deviation of differences between consecutive dates to measure the uniformity of date distribution:

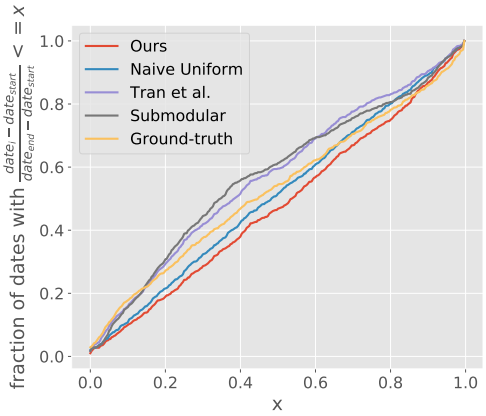
Definition 3 (Uniformity of Date Selection). *Given a series of selected dates $\{d_1, d_2, ..., d_T\}$ in chronological order, we regard the differences between consecutive dates as $\{diff_i = d_{i+1} - d_i\}$, then define its standard deviation $\sigma = \sqrt{\frac{1}{T}\sum_{i=1}^{T}(diff_i - \bar{diff})^2}$, as the uniformity of date selection.*

2.2.1 Recency Adjustment. To add more weights on recent dates, we leverage the Personalized PageRank algorithm [1], where the restart distribution is not uniform. More specifically, we weight each date node $date_i$ by $W_i = \alpha^{-d_i}$, where $d_i = |date_i - date_{start}|$. α ranges from 0 to 1 and is used to control the restart distribution. In practice, we use a grid search to find the α that gives the most uniform distribution in the date selection, then use the chosen dates for news timeline generation.

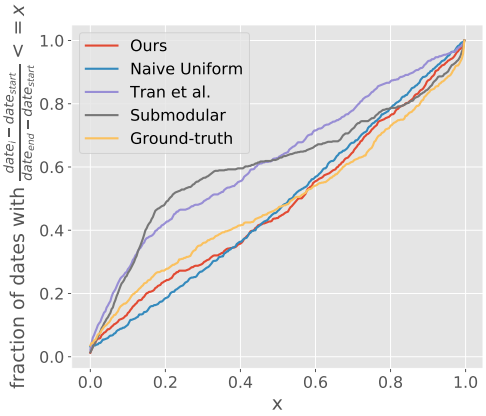
2.2.2 Date Coverage. To better check the coverage of generated timelines, besides f1 score on date selection, we also measure the date coverage, e.g., if any day of ground-truth date $g_i \pm 3$ days lies in the generated timeline, g_i will be considered to be covered and we will measure what percentage of ground-truth dates are covered per timeline. For comparison, we also generate news timelines on truly uniformly distributed dates and present the results in Table 3. As we can see, although truly uniformly distributed dates cover the most ground-truth dates, due to the low accuracy in the date selection, the generated daily summaries are poor. However, adding recency adjustment with uniformity contributes to date selection in coverage, thus yields better timeline summarization.

2.3 Daily Summarization

Having selected the most salient dates, next, we divide timeline summarization into sub-summarization tasks. Although daily summarization tasks can be accomplished by any supervised or unsupervised document summarization algorithms, we intend to use a simple daily summarization method to validate the effectiveness of our explicit date selection, as complicated summarization techniques may introduce extra improvements in the performance. Specifically, we utilize the classic TextRank



(a) Timeline17



(b) Crisis

Figure 4: Distribution of selected dates among different approaches.

[13] to generate daily summaries. Similar to the task of date selection, TextRank constructs a sentence graph with sentences as nodes and similarity scores as edge weights. In particular, we use BM25 [18] to compute edge weights [2] and run PageRank on this directed graph to select the most important sentences as daily summaries.

2.3.1 Post-processing. Dividing large text summarization tasks into smaller ones greatly speeds up timeline generation, and these sub-tasks can naturally be further accelerated through parallel processing. Conducting text summarization on a daily basis rather than on the whole corpus, however, ignores temporal correlation and thus introduces redundancy in summarization. To remove redundancy across dates, therefore, we incorporate post-processing to re-rank daily summaries based on the whole summarization. Similar to MMR [3], instead of directly using daily summaries, we add sentences into timeline summarization by their daily ranks and only accept sentences whose maximum cosine similarity with selected ones is smaller than a threshold (e.g., < 0.5).

2.4 News Timeline Generation Algorithm

The generation algorithm of WILSON is summarized in Algorithm 1. First, we build a date reference graph based on the date

Algorithm 1: Algorithm for WILSON

Input : temporally tagged sentences $C = \{(date_i, sent_i)\}$
preset number of dates T
preset number of daily sentences N

Output : a series of daily summaries $(d_1, S_1), ..., (d_T, S_T)$

Build a date reference graph based on date co-occurrence $\{(date_i, date_j) | (date_i, sent_k) \in C \ \& \ (date_j, sent_k) \in C\}$;
Compute edge weight according to W3 in Section 2.2 ;
selected_dates $\leftarrow \emptyset$;
for *Grid search* $\alpha \in (0, 1)$ **do**
    Compute personalized node weight for each date $date_i$ *using* $W_i = \alpha^{-|date_i - \min_k(date_k)|}$;
    Run personalized PageRank to select the top T *ranked dates as a date selection candidate ;*
    Based on Definition 3, compute the uniformity score of this date selection candidate ;
    selected_dates $\leftarrow$ *save the date selection with the best uniformity score as* $(d_1, d_2, ..., d_T)$;
end for
for $d_i \in$ *selected_dates* **do**
    Find all sentences on d_i
    $C_i \leftarrow \{sent_k | (date_k, sent_k) \in C \ \& \ date_k = d_i\}$;
    Run TextRank on C_i *to rank all sentences by importance score in a max heap* H_i ;
    Initialize selected sentences $S_i \leftarrow \emptyset$;
end for
repeat
    Currently selected sentences $S \leftarrow \bigcup_{i=1}^{T} S_i$;
    Top ranked sentence per day $H \leftarrow \bigcup_{i=1}^{T} H_i[0]$;
    Remove top sentences: heap_pop(H_i*) for* $i \in$ *[1, T];*
    Remove sentences from H *that have maximum similarity > 0.5 with existing sentences in* S ;
    Add remaining sentences in H *to the corresponding daily summary* S_i *only if* $|S_i| < N$;
until *(all* $|S_i| = N$*) or (all* $|H_i| = 0$*);*
return $(d_1, S_1), ..., (d_T, S_T)$

pairs that appear in the same sentences. Second, we extract features to compute weights for the graph edges and run personalized PageRank to pick the most salient T dates. More specifically, we include the recency adjustment strategy to improve the date coverage of selected dates. Then, we use TextRank to rank all sentences on each selected date. According to the sentence ranks per selected date, we post-process the sentences in batch and remove sentences that could introduce redundant information on existing selections. Finally, our algorithm produces a series of compact daily briefs as the summarized news timeline to help people better understand the evolution of the corresponding news event.

2.5 Complexity Analysis

In this section, we briefly provide a time complexity analysis of our approach with a comparison to the submodular framework. Denote T as the total number of dates, N as the average number of sentences per date, t as the desired number of dates and n as the desired number of sentences per date in the summarized timeline. According to PageRank on the dense graph, date selection takes $O(T^2)$ while t daily summarization tasks take $O(t * N^2)$. Thus the total time complexity of WILSON is $O(T^2 + t * N^2)$.

For submodular framework [12], which conducts global summarization, it takes $O((TN)^2)$ to obtain pair-wise similarities for all sentences and takes $O(t * n * T * N)$ to iterate $t * n$ times to select each individual sentence in a greedy manner. Therefore, the total time complexity is $O((TN)^2 + t * n * N * T)$. In Figure 2, the corpus size is defined as the total number of sentences (i.e. $T * N$). As expected, the submodular frameworks show quadratic growth with a time complexity $O((TN)^2)$, while our approach is almost linear to the corpus size with a time complexity $O(T^2 + t * N^2)$.

Given the approximation that T and N are in the same order of magnitude (based on Table 4), WILSON runs faster than the submodular framework by a factor of $O(\frac{T^2}{t})$. Given around 10% date compression rate ($\frac{t}{T}$) and T in hundred level, theoretically, our approach could gain over three orders of magnitude improvement in generation speed. Note that, due to the scalability issue of the submodular framework, [12] filtered sentences with predefined keywords to reduce N by over one order of magnitude, reducing the time complexity in practice. Given ~10% filtering rate, our approach could still gain about two orders of magnitude in generation speed, which is consistent with experiments in Table 7.

Dataset	# of topics	# of timelines	average per timeline		
			# of doc	# of sents	duration days
Timeline17	9	19	739	36,915	242
Crisis	4	22	5,130	173,761	388

Table 4: Dataset overview

3 EMPIRICAL VALIDATION

3.1 Set-Up

3.1.1 Datasets. We run experiments on *timeline17* [24, 25] and *crisis* [22]. Both datasets[2] consist of journalist generated timelines from major news media such as CNN, BBC and Reuters, and a corresponding corpus of articles per topic (e.g. H1N1 flu and Egypt war). More specifically, *timeline17* contains 19 timelines from 9 topics, while *crisis* involves 22 timelines from 4 topics. An overview of the two datasets is shown at Table 4.

3.1.2 Competing methods.

- **Random**: The system generates daily summaries by randomly selecting sentences from the corpus.
- **MEAD** [17]: a classic centroid-based multi-document summarization system.
- **Chieu et al.** [4]: a multi-document summarization system that uses date related TFIDF scores to measure sentence importance among corpus.
- **ETS** [29]: an unsupervised timeline summarization algorithm via simultaneously optimizing multiple heuristic metrics, including relevance, coverage, coherence, and diversity.
- **Tran et al.** [25]: a supervised timeline summarization algorithm, which extracts various features from sentences and leverages learning to rank techniques.
- **Regression** [26]: a supervised approach that formulates sentence selection as a linear regression problem.
- **Wang et al.** [27]: a supervised approach that formulates sentence selection as a matrix factorization problem.

[2] http://l3s.de/~gtran/timeline/

Methods	ROUGE-1	ROUGE-2	ROUGE-S*
Random	0.128	0.021	0.026
Chieu et al.	0.202	0.037	0.041
MEAD	0.208	0.049	0.039
ETS	0.207	0.047	0.042
Tran et al.	0.230	0.053	0.050
Regression	0.303	0.078	0.081
Wang et al. (Text)	0.312	0.089	0.112
Wang et al. (Text + Vision)	0.331	0.091	0.115
Liang et al.	0.334	**0.105**	0.103
WILSON **(Ours)**	**0.370**	0.083	**0.141**

Table 5: Results on Timeline17

- **Liang et al.** [9]: a dynamic evolutionary framework leveraging distributed representation for timeline summarization.
- **ASMDS (TILSE)** [12]: TILSE is a state-of-the-art unsupervised timeline summarization approach, which incorporates submodularity-based multi-document summarization framework with temporal criteria.
- **TLSCONSTRAINTS (TILSE)** [12]: as a variant of TILSE, this method uses the same objective funtion as ASMDS but adopt different temporal constraints.

3.1.3 Measurement. Among all the baselines, TILSE is the only one with source code available. Consequently, for all the other baselines, we follow the existing works [9, 25, 27], which conduct experiments on *timeline*17 with settings mentioned at the beginning of Section 5.2 in [25] and directly report the baseline results from previous papers. More specifically, in the generated timeline, the number of selected dates T is set to the number of dates in each ground-truth timeline, while the number of sentences per day N is forced to be the rounded value of the average number of sentences per date from the ground-truth timeline.

To fairly compare with TILSE, we re-run the their code, follow all their pre-processing, including text cleaning and keywords filtering, and conduct experiments on exactly the same sentence corpus per timeline generation. Note that, [12] used a slightly different setting from previous papers: 1) for Timeline17 dataset, they mixed articles of the same topic from different news agencies together, which yields a bit higher ROUGE scores in timeline generation; 2) it suffers from the scalability issue and thereby uses filtered sentence corpus for both datasets. Thus, we followed their settings for a fair comparison with TILSE in Table 7. Wall time is measured on a 24-core 128GB machine.

3.1.4 Evaluation Metrics. The commonly used summarization metrics, ROUGE scores [10], including ROUGE-1, ROUGE-2 and ROUGE-S* F1 scores, are adopted to evaluate the agreement between machine-generated and journalist generated timelines. Moreover, to be consistent with TILSE comparison, we also include time-sensitive ROUGE scores as additional measurements [11]. More specifically, *concat* ROUGE scores totally ignore the time information by directly concatenating all texts together, while *agreement* ROUGE scores only consider the generated daily summaries on the groundtruth dates, and *align* ROUGE scores discount the quality of generated daily summaries by their distance to the corresponding groundtruth date. Last but not least, we test for significant improvements using an approximate randomization test [15] with a p-value of 0.05.

Methods	ROUGE-1	ROUGE-2	ROUGE-S*
Regression	0.207	0.045	0.039
Wang et al. (Text)	0.211	0.046	0.040
Wang et al. (Text + Vision)	0.232	0.052	0.044
Liang et al.	0.268	0.057	0.054
WILSON **(Ours)**	**0.352**	**0.074**	**0.123**

Table 6: Results on Crisis

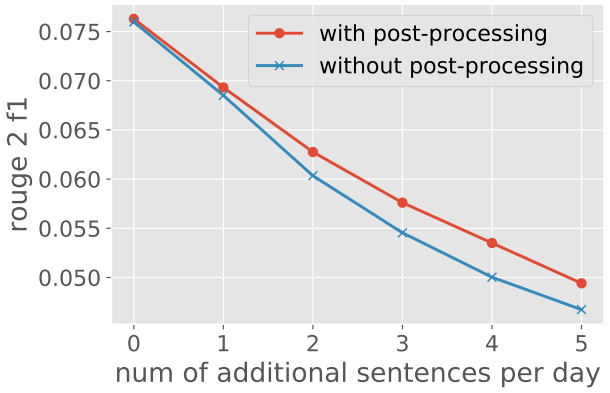


Figure 5: Concat Rouge 2 f1 scores when adding more sentences on each date on Crisis.

3.2 Performance Comparison

Table 5 and 6 shows that our unsupervised approach WILSON outperforms all baselines in ROUGE-1 and ROUGE-S* f1 scores by a significant margin, and is only second to [27], a supervised approach, and [9] in ROUGE-2 f1 score on Timeline17 dataset.

In addition, Table 7 illustrates that WILSON outperforms the state-of-the-art unsupervised framework TILSE in all ROUGE metrics. Averagely, our method outperforms the submodular approaches by 12.9% in concatenate ROUGE-2 scores, by 58.3% in agreement ROUGE-2 scores, and by 40.1% in alignment ROUGE-2 scores. More importantly, our method also gains two orders of magnitude improvement in generation speed, making it possible to generate news timelines in a real-time manner.

In Table 7, We also include multiple variants of WILSON for ablation analysis. WILSON-uniform simply adopts uniform date selection, while WILSON-Tran directly uses W3 as edge weight without recency adjustment. As expected, selecting uniformly distributed dates results in the worst summarization, while including recency adjustment improves time-sensitive ROUGE-2 scores by 9.0%~21.6%.

Overall, comparing with all competing approaches, the performance improvement of our method is higher in *Crisis* dataset. One explanation is that *Crisis* dataset contains more articles and spans a longer period, making it difficult for those competing approaches to correctly identify the long-term event dependencies, while our method mainly focuses on local dependencies.

3.2.1 Effectiveness of Post-processing. In Table 7, we observed that considering correlation across different dates and reducing redundant daily summaries are seemingly minor, especially on *Crisis* datasets. Different from *Timeline17* datasets, *Crisis* datasets consist of more compact daily summaries, where more than 90% dates contain only 1 sentence. Although reducing redundancy across dates is not necessary for timelines with compact daily summaries, we intend to verify the effectiveness of

Model	concat Rouge 1	concat Rouge 2	concat our impr.	agreement Rouge 1	agreement Rouge 2	agreement our impr.	align+ m:1 Rouge 1	align+ m:1 Rouge 2	align+ m:1 our impr.	Date F1	Running Time Per timeline (sec.)
Timeline17											
ASMDS	0.3452	0.0890	13.8%	0.0913	0.0270	20.0%	0.1047	0.0299	17.1%	0.5437	338.68
TLSCONSTRAINTS	0.3685	0.0916	10.6%	0.0912	0.0242	33.9%	0.1049	0.0270	29.6%	0.5127	560.24
WILSON-uniform	0.3659	0.0848	19.5%	0.0754	0.0191	69.6%	0.0924	0.0218	60.6%	0.4366	**1.97**
WILSON-Tran	0.4007	0.0993	2.0%	0.1035	0.0293	10.6%	0.1181	0.0321	9.0%	**0.5668**	2.12
WILSON w/o Post	0.4036	0.1005	0.8%	0.1057	0.0318	1.9%	0.1202	0.0344	1.7%	0.5542	5.63
WILSON	**0.4075**⋆†	**0.1013**⋆†		**0.1065**⋆†	**0.0324**†		**0.1211**⋆†	**0.0350**†		0.5542	7.59
Crisis											
ASMDS	0.3066	0.0645	17.7%	0.0415	0.0091	123.1%	0.0658	0.0135	71.9%	0.2435	3055.96
TLSCONSTRAINTS	0.3307	0.0693*	9.5%	0.0564	0.0130	56.2%	0.0764	0.0166	39.8%	0.2739	4098.07
WILSON-uniform	0.3314	0.0551	37.7%	0.0235	0.0059	244.1%	0.0392	0.0080	190.0%	0.1251	**4.68**
WILSON-Tran	0.3575	0.0739	2.7%	0.0621	0.0167	21.6%	0.0798	0.0202	14.9%	0.2726	5.69
WILSON w/o Post	0.3600	0.0756	0.4%	0.0677	0.0201	1.0%	0.0843	0.0230	0.9%	**0.2748**	22.95
WILSON	**0.3605**⋆†	**0.0759**⋆†		**0.0679**⋆	**0.0203**⋆†		**0.0846**⋆	**0.0232**⋆		**0.2748**	30.14

Table 7: Comparison with TILSE. We indicate our improvement on Rouge 2 f1 score for different metrics. For WILSON**, we use an approximate randomization test to test the significance of its improvement over ASMDS and TLSCONSTRAINTS, and denote the significant improvement by ⋆ and † respectively.**

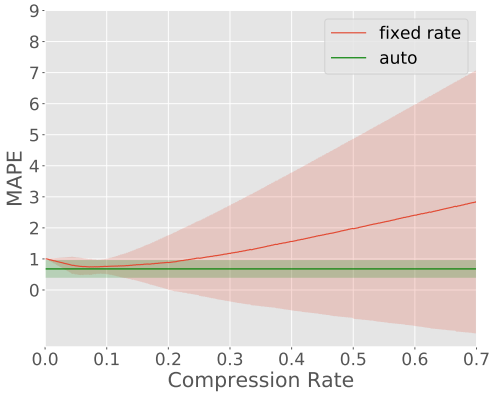
Method	ROUGE-1	ROUGE-2
timeline17		
Submodularity framework [12]	0.50	0.18
Ground-truth date + Daily summary	0.41	0.11
Crisis		
Submodularity framework [12]	0.49	0.16
Ground-truth date + Daily summary	0.42	0.10

Table 8: Empirical upper bound of submodularity framework and our two-stage method

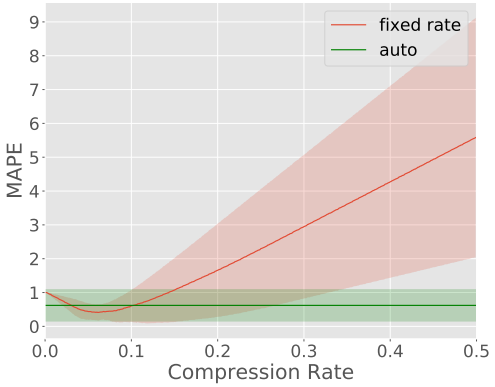
post-processing for timelines with abundant daily summaries. Instead of using the exact number of sentences per date in the ground-truth timelines, we generate timelines with more sentences per date, which is more practical as the true numbers are unknown. As demonstrated in Figure 5, simply adding daily summaries together suffers from the redundancy issue and using post-processing indeed helps. Note that we use the ROUGE-2 f1 score, so the overall scores going down with more sentences is because more generated texts lead to lower ROUGE accuracy.

3.2.2 Empirical Bounds. Empirical bounds of our two-stage method are given in Table 8, where we use ground-truth dates as date selections for daily summarization. Note that, besides using ground-truth dates, the upper bounds of the submodularity framework [12] also employ ground-truth summaries and are obtained by directly optimizing ROUGE f1 scores in a supervised way, but we only use ground-truth dates and never touch ground-truth summaries, making us aware of how date selection will contribute to news timeline summarization. As demonstrated, even without considering contextual correlation across different dates in text summarization, it is still possible to generate reasonable news timelines with accurate date selection. Although the upper bound of our two-stage framework is much lower than that of the submodular framework, it is worth mentioning that all existing approaches fail to reach our upper bound, not even close on the *Crisis* dataset.

3.2.3 Automatic Date Compression. As defined in Section 2.1, existing news timeline summarization works only use a preset number of dates and length of daily summaries to generate news timelines. Unlike the length of daily summaries, which only implies the compression rate for a single day and is usually set



(a) Timeline17



(b) Crisis

Figure 6: Mean Absolute Percentage Error (MAPE) of predicted number of date selection

to 2 or 3 sentences, determining the number of dates requires understanding for the whole corpus, making it difficult to select. To solve this issue, we aim at automatically detecting the number of dates for news timelines. Motivated by the fact that news timelines consist of major events within the duration, we propose to consider major event coverage to determine the number of dates. Specifically, we use the daily summarization to generate major events for each date and encode daily summaries with BERT

Method	rank 1st	2nd	3rd	MRR	DCG
ASMDS	4	3	3	0.72	7.39
TLSCONSTRINTS	1	6	3	0.56	6.29
WILSON (Ours)	**5**	1	4	**0.76**	**7.63**

Table 9: Results of journalist evaluation on the quality of machine-generated timelines. Best and second best scores are highlighted by bold and underscore respectively.

[5] into embedding vectors. Then, we use Affinity Propagation [6] to cluster encoded daily summaries into event clusters, and adopt the detected cluster number as date selection number. We compared our methods with fixed compression rates for date selection and presented the results in Figure 6. As shown, our automatic date compression method generally performs well on both datasets.

3.3 Evaluation by Journalists

In addition to ROUGE scores, we also consult two professional journalists at *the Washington Post*, which is one of the leading daily American newspapers, to manually evaluate the quality of machine-generated news timelines. Among 41 timelines from the two datasets, we sample 10 timelines (20%) from 6 topics, including H1N1 flu, BP oil spill, Egypt crisis, Libya war, Yemen war, and Syria war. For each sampled timeline, we present the human-generated ground-truth timeline and three machine-generated timelines from ASMDS, TLSCONSTRINTS, and WILSON (Ours) to journalists. The ground-truth timeline is labeled as a reference, while the other three are given in random order and the order is independent for each evaluation. The evaluation is based on the comprehensiveness and readability of the generated timelines compared with the ground-truth timelines.

For each evaluation, the two journalists are asked to review ~ 80 daily summaries from ~ 50 distinct dates, which adds up to ~ 800 daily summaries from ~ 500 distinct dates in total, and collaborate to provide one final ranking of the three machine-generated timelines. To measure the ranking performance of each method, we adopt two common rank-aware measurements, Mean Reciprocal Rank (MRR) and Discounted Cumulative Gain (DCG), and present the results in Table 9. As shown, when evaluated by professional journalists, our method outperforms the state-of-the-art unsupervised framework TLSCONSTRAINS and achieves slightly better or comparable results with ASMDS. Considering our method gains two orders of magnitude improvement in generation speed, the results are very encouraging. More interestingly, although TLSCONSTRAINS generally achieves higher ROUGE scores than ASMDS in table 7, TLSCONSTRAINS receives unexpectedly lower rank scores than ASMDS in this evaluation by journalists. This may imply a warning that automated measures may not be enough for news timeline summarization and human evaluation could be beneficial at times.

4 A CASE STUDY

In this section, we perform a qualitative analysis of the generated timelines of our approach. Since TILSE [12] is the only baseline with source code available, we also include its output in comparison. Table 10 presents a subset of timelines about the lawsuit of Michael Jackson's death in the *Timeline17* dataset, where the manually generated timeline was collected from BBC[3]. As different approaches generate timelines with different date selections, we only consider the dates that appear in all 4 timelines and show the first a few dates and their summaries in chronological order. We highlight the overlaps between manually generated and automatic generated timelines in colors and observe that the output of our approach is aligned better with the handcrafted one.

Interestingly, more summaries of our outputs are closer to the main events on each date than those of TILSE's, though they are all relevant to this topic. We think it may be because more important daily events are reported more heavily on each date, while existing models try but fail to effectively capture the evolution clues across dates, thus simple daily summarization can work well. Apparently, how to balance local and global summarization and effectively capture event evolution could be one potential direction for news timeline summarization.

5 REAL-TIME SYSTEM FRAMEWORK

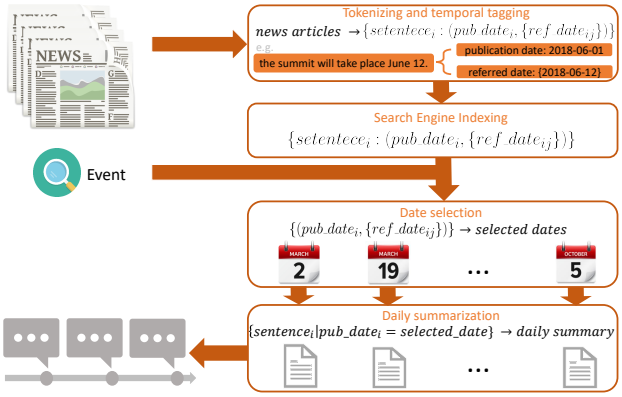


Figure 7: Framework for Real-Time News Timeline Summarization

The framework of our real-time news timeline generation system is shown in Figure 7. This framework applies our proposed method WILSON on a 4-year news corpus of over 1 million news articles[4] from the Washington Post and can generate timelines by event keywords in seconds. Firstly, we tokenize all the news articles into sentences and conduct temporal tagging to label each sentence. Then, to query relevant news content in real-time, we build a search engine on tagged sentences and index both date and content information. Specifically, we use ElasticSearch [7] as our backend search engine. Note that, we can easily include newly published news articles into our system by inserting them into the existing search engine. Finally, given both the keywords and duration time of a query event, our system will fetch related news sentences and run WILSON to generate a complete news timeline.

For example, we can generate a timeline about how the United States and North Korea reached the summit in seconds by setting query keywords to "trump, north korea, kim, summit, united states" and time duration between 2018-01-02 to 2018-06-12. We set the timeline length to 10 and present the output in Table 11. Taking journalist generated timeline[5] as a reference, we highlight

[3]https://www.bbc.com/news/entertainment-arts-15060651
[4]We excluded all news articles containing the keyword "timeline".
[5]https://www.washingtonpost.com/graphics/2018/national/trump-kim-jong-un-timeline/

Groundtruth (From BBC)	TILSE (TLSCONSTRAINTS)	TILSE (ASMDS)	WILSON (Ours)
2009-06-25 Dr Murray finds Jackson unconscious in the bedroom of his Los Angeles mansion . Paramedics are called to the house while Dr Murray is performing CPR , according to a recording of the 911 emergency call . He travels with the singer in an ambulance to UCLA medical center where Jackson later dies .	2009-06-25 Jackson died at his Los Angeles home on 25 June aged 50 . Jackson died at his home on 25 June last year at the age of 50 .	2009-06-25 Michael Jackson died on 25 June 2009 from an overdose of the powerful anesthetic propofol .	2009-06-25 Same drug class as morphine Given by tablets or injection Used post-surgery or for childbirth High doses can stop breathing or lead to delirium and seizures Jackson , who had a history of health problems , collapsed at his Los Angeles home around midday on Thursday . Mr Martinez , who interviewed Dr Murray two days after Jackson 's death on 25 June 2009 , said the doctor told him the singer had stopped breathing shortly after 1100 .
2009-06-28 Los Angeles police interview Dr Murray for three hours . His spokeswoman insists he is " not a suspect " .	2009-06-28 Jackson 's body was released to the family on Friday night . Jackson 's body was released to the family on Friday night .	2009-06-28 Jackson family left ' speechless and devastated ' by star 's death Relatives of Michael Jackson will seek a second autopsy on the star because they still have unanswered questions about his death , family friends say .	2009-06-28 Michael Jackson 's family are said to be seeking a second autopsy because they still have questions about his death . Earlier , veteran politician Rev Jesse Jackson , who has been counselling the family , said they had a flurry of questions of their own for Dr Murray .
2009-07-28 Dr Murray 's home is also raided . The search warrant allows " authorised investigators to look for medical records relating to Michael Jackson and all of his reported aliases " . A computer hard drive and mobile phones are seized , and a pharmacy in Las Vegas is later raided in connection with the case .	2009-07-28 Dr Conrad Murray , who police say is not a suspect , was at Jackson 's mansion and tried to revive him before he died . Police raid Jackson doctor 's home Drug police are searching the Las Vegas home of Michael Jackson 's doctor as part of a manslaughter investigation into the singer 's death .	2009-07-28 On Tuesday , police searched the Las Vegas home and offices of Jackson 's doctor , Conrad Murray , as part of a manslaughter investigation into the singer 's death .	2009-07-28 Police raid Jackson doctor 's home Drug police are searching the Las Vegas home of Michael Jackson 's doctor as part of a manslaughter investigation into the singer 's death . On Tuesday , police searched the Las Vegas home and offices of Jackson 's doctor , Conrad Murray , as part of a manslaughter investigation into the singer 's death .
2010-06-25 Michael Jackson 's father , Joseph , files a wrongful death lawsuit against the physician .	2010-06-25 Randy Jackson recently succeeded in stopping an unapproved tribute show to his brother Michael in Rome , which had been scheduled for 25 June , the anniversary of his death . The suit was filed as fans around the world marked the first anniversary of Jackson 's death at the age of 50 .	2010-06-25 Jackson died of a cardiac arrest at his home on 25 June last year .	2010-06-25 25 June 2010 Michael Jackson 's father , Joseph , files a wrongful death lawsuit against the physician . Fans sing outside the Jackson family home .
2011-07-25 Rehearsal footage from Michael Jackson 's This Is It tour can not be used as evidence , the judge rules .	2011-07-25 Judge Michael Pastor concluded on Monday that it would not help the defense and that " it was a waste of my time . " 25 July 2011 Rehearsal footage from Michael Jackson 's	2011-07-25 But Judge Michael Pastor ruled on Monday that the film would not help the defense team and was a waste of his time .	2011-07-25 Judge Michael Pastor concluded on Monday that it would not help the defense and that " it was a waste of my time . 25 July 2011 Rehearsal footage from Michael Jackson 's
2011-08-30 Michael Jackson 's dermatologist is barred from giving evidence at the trial . Dr Murray 's lawyers had planned to argue that Arnold Klein had administered the singer with painkillers for " no valid reason " but prosecutors said they were attempting to transfer responsibility for his death away from Dr Murray . Testimony from five other doctors who treated Jackson is also disallowed .	2011-08-30 Janet Jackson to miss concert Janet Jackson said she would find it " difficult " to attend the tribute concert in Cardiff Janet Jackson will not be attending her brother Michael Jackson 's tribute concert in Cardiff . Because of the trial , the timing of this tribute to our brother would be too difficult for me , " Ms Jackson said in a statement .	2011-08-30 Janet Jackson to miss concert Janet Jackson said she would find it " difficult " to attend the tribute concert in Cardiff Janet Jackson will not be attending her brother Michael Jackson 's tribute concert in Cardiff .	2011-08-30 But Superior Court Judge Michael Pastor ruled that Arnold Klein would not be called to testify after prosecution lawyers said the defense wanted to transfer responsibility for Jackson 's death to the dermatologist . Because of the trial , the timing of this tribute to our brother would be too difficult for me , " Ms Jackson said in a statement .
2011-09-29 Jackson 's bodyguard , Alberto Alvarez , testifies that on the night Jackson died , Dr Murray ordered him to pick up vials of medicine before phoning for an ambulance . " In my personal experience , I believed Dr Murray had the best intentions for Mr Jackson , " Mr Alvarez said .	2011-09-29 29 September 2011 Last updated at 15:44 GMT Help Live coverage of the trial of Michael Jackson 's personal physician , Dr Conrad Murray , who is charged with involuntary manslaughter of the singer . 29 September 2011 Last updated at 04:16 GMT Help A key aide and a security guard have told the manslaughter trial of Michael Jackson 's doctor of events on the day the superstar died .	2011-09-29 However , Jermaine and Randy Jackson said it should not go ahead because it would clash with the trial of Conrad Murray , the singer 's former doctor accused of his involuntary manslaughter .	2011-09-29 Jackson 's bodyguard Alberto Alvarez claims Dr Murray " grabbed a handful of vials " and told him to put them in a bag Michael Jackson 's doctor told the performer 's bodyguard to pick up vials of medicine before phoning for help on the day he died , his trial has heard . 29 September 2011 Last updated at 04:16 GMT Help A key aide and a security guard have told the manslaughter trial of Michael Jackson 's doctor of events on the day the superstar died .

Table 10: Summary examples on the Death of Michael Jackson. To save space, we only select the first a few dates in chronological order, which appear in all 4 timelines. Main coverage with groundtruth is colored: red texts highlight the overlaps between groundtruth and TILSE/ours while blue texts highlight the distinct overlaps between groundtruth and ours. Note that all summarization approaches use exactly the same sentence candidates pool.

2018-03-08 Jason Aldag The Post reports : North Korea 's belligerent leader , Kim Jong Un , has asked President Trump for talks and Trump has agreed to meet him " by May , ...	2018-04-01 CIA Director Mike Pompeo , left , and North Korean leader Kim Jong Un shake hands during a meeting in in Pyongyang , North Korea on Easter Weekend .
2018-04-16 The only way the United States can persuade North Korea to peacefully give up its pursuit of these weapons is if Kim believes Trump 's threat of military force is credible .	2018-04-20 7:30 a.m. Friday North Korea 's state media reports that leader Kim Jong Un has left Pyongyang for the North - South summit meeting with South Korean President Moon Jae - in .
2018-04-27 ... North Korean leader Kim Jong Un on Friday morning ... walking into South Korea for a historic summit with President Moon Jae - in that will lay the groundwork for a meeting between Kim and President Trump .	2018-05-09 Their release came as Secretary of State Mike Pompeo visited North Korea on Wednesday to finalize plans for a historic summit meeting between Trump and the North 's leader , Kim Jong Un .
2018-05-16 North Korea has taken repeated ... and threatening to scrap next month 's planned summit between Kim and U.S. President Donald Trump , saying it won't be unilaterally pressured into relinquishing its nuclear weapons .	2018-05-24 After weeks of receiving and even appearing to encourage chants of " Nobel " ahead of a planned historic meeting with North Korea dictator Kim Jong Un , President Trump on Thursday abruptly canceled the June 12 summit .
2018-06-05 ... Donald Trump cast his Tuesday summit with North Korea 's Kim Jong Un as a " one - time shot " for the autocratic leader to ditch his nuclear weapons and enter the community of nations ...	2018-06-12 President Trump said the U.S. will end its " war games " with South Korea after the historic summit with North Korean leader Kim Jong Un on June 12 .

Table 11: WILSON **generated output of the timeline about how the U.S. and North Korea finally had the summit. Main coverage with journalist generated timeline is color-coded blue, and some trivial contexts are omitted by ellipsis for space.**

the coverage between our generated news timeline with the journalist-generated timeline in blue color and demonstrate that our output is aligned well with the journalist-generated timeline, showing the effectiveness of WILSON in practice.

6 RELATED WORK

Existing works in summarizing timelines for a specific topic from relevant news articles include both supervised and unsupervised approaches. As representatives of supervised approaches, [25] leverages learning to rank techniques based on sentence features, while [27] proposes a matrix factorization framework to predict importance scores of sentences. Unsupervised approaches usually optimize task-specific heuristic object functions, which measure relevance, coverage and diversity of daily summaries. For example, [28] solves the optimization problem by iteratively substituting sentences in summaries, while the state-of-the-art framework TILSE adapts the sub-modularity framework from multi-document summarization domains to optimize timeline summarization [12].

In addition to extractive methods, some recent works also utilize abstractive summarization methods to generate more compact sentences as daily summaries [19]. Although the generated sentences are empirically proved to be readable, the reliability of generated summaries are not guaranteed, probably leading to false information. Extractive summarization methods, however, directly borrow sentences from original news articles and do not encounter reliability issue. Thus, in this paper, we utilize extractive summarization for both readability and reliability of generated timelines.

Besides ROUGE scores, [19] is the only existing work to include human in the evaluation, but they just assess the readability of daily summaries as they utilize the abstractive summarization. Since none of the previous studies utilize user study to measure the generation quality of the whole news timelines, we are the first work to include user study in timeline evaluation and consult journalists to assess the generation quality of the entire news timelines.

7 CONCLUSION

This paper shows that, with accurate date selection, we can generate high-quality news timelines without considering the temporal correlation of text summarization. Leveraging the explicit date selection, we propose a fast and effective unsupervised timeline summarization method named WILSON. Specifically, WILSON outperforms state-of-the-art approaches in both ROUGE scores and speed, significantly improving concatenate ROUGE-2 F1 scores by 9.5%~17.7%, time-sensitive ROUGE-2 F1 scores by 17.1%~123.1% and reducing generation time by two orders of magnitude, which allows us to develop a real-time news timeline generation system for the news room. More importantly, a user study with professional journalists also confirms that the outputs of WILSON are closer to human-generated ones than outputs of other methods. Last but not least, this work also suggests two potential directions for future works, i.e. considering both occurrence and recency of events for better salient date selection and reducing contextual correlation across dates by balancing local and global summarization to improve daily summarization.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work was in part supported by NSF awards #1742702, #1820609, #1909702, #1915801, and #1934782. Any opinions, findings and conclusions or recommendations expressed in this material are the author(s) and do not necessarily reflect those of the sponsors. Last but not least, we truly appreciate Everdeen Mason, Sophie Ho and their journalist team at the Washington Post for the valuable feedback and support.

REFERENCES

[1] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. 2010. Fast incremental and personalized pagerank. *Proceedings of the VLDB Endowment* 4, 3 (2010), 173–184.
[2] Federico Barrios, Federico López, Luis Argerich, and Rosa Wachenchauzer. 2016. Variations of the similarity function of textrank for automated summarization. *arXiv preprint arXiv:1602.03606* (2016).
[3] Jaime Carbonell and Jade Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 335–336.
[4] Hai Leong Chieu and Yoong Keok Lee. 2004. Query based event extraction along a timeline. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 425–432.
[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[6] Brendan J Frey and Delbert Dueck. 2007. Clustering by passing messages between data points. *science* 315, 5814 (2007), 972–976.
[7] Clinton Gormley and Zachary Tong. 2015. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.".
[8] Jiwei Li and Claire Cardie. 2014. Timeline generation: Tracking individuals on twitter. In *Proceedings of the 23rd international conference on World wide web*. ACM, 643–652.
[9] Dongyun Liang, Guohua Wang, and Jing Nie. 2019. A Dynamic Evolutionary Framework for Timeline Generation based on Distributed Representations. *arXiv preprint arXiv:1905.05550* (2019).

[10] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. *Text Summarization Branches Out* (2004).
[11] Sebastian Martschat and Katja Markert. 2017. Improving rouge for timeline summarization. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. 285–290.
[12] Sebastian Martschat and Katja Markert. 2018. A Temporally Sensitive Submodularity Framework for Timeline Summarization. In *Proceedings of the 22nd Conference on Computational Natural Language Learning*. 230–240.
[13] Rada Mihalcea and Paul Tarau. 2004. Textrank: Bringing order into text. In *Proceedings of the 2004 conference on empirical methods in natural language processing*.
[14] Jun Ping Ng, Yan Chen, Min-Yen Kan, and Zhoujun Li. 2014. Exploiting timelines to enhance multi-document summarization. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 923–933.
[15] Eric W Noreen. 1989. *Computer-intensive methods for testing hypotheses*. Wiley New York.
[16] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The pagerank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.
[17] Dragomir R Radev, Hongyan Jing, Małgorzata Styś, and Daniel Tam. 2004. Centroid-based summarization of multiple documents. *Information Processing & Management* 40, 6 (2004), 919–938.
[18] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.
[19] Julius Steen and Katja Markert. 2019. Abstractive Timeline Summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*. 21–31.
[20] Jannik Strötgen and Michael Gertz. 2015. A Baseline Temporal Tagger for all Languages. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Lisbon, Portugal, 541–547. http://aclweb.org/anthology/D15-1063
[21] Russell C Swan and James Allan. 2000. TimeMine: visualizing automatically constructed timelines.. In *SIGIR*. 393.
[22] Giang Tran, Mohammad Alrifai, and Eelco Herder. 2015. Timeline summarization from relevant headlines. In *European Conference on Information Retrieval*. Springer, 245–256.
[23] Giang Tran, Eelco Herder, and Katja Markert. 2015. Joint graphical models for date selection in timeline summarization. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics*, Vol. 1. Association for Computational Linguistics, 1598–1607.
[24] Giang Binh Tran, Mohammad Alrifai, and Dat Quoc Nguyen. 2013. Predicting relevant news events for timeline summaries.. In *WWW (Companion Volume)*. 91–92.
[25] Giang Binh Tran, Tuan A Tran, Nam-Khanh Tran, Mohammad Alrifai, and Nattiya Kanhabua. 2013. Leveraging learning to rank in an optimization framework for timeline summarization. In *SIGIR 2013 Workshop on Time-aware Information Access (TAIA*.
[26] Lu Wang, Claire Cardie, and Galen Marchetti. 2015. Socially-Informed Timeline Generation for Complex Events. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 1055–1065.
[27] William Yang Wang, Yashar Mehdad, Dragomir R Radev, and Amanda Stent. 2016. A low-rank approximation approach to learning joint embeddings of news stories and images for timeline summarization. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 58–68.
[28] Rui Yan, Liang Kong, Congrui Huang, Xiaojun Wan, Xiaoming Li, and Yan Zhang. 2011. Timeline generation through evolutionary trans-temporal summarization. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 433–443.
[29] Rui Yan, Xiaojun Wan, Jahna Otterbacher, Liang Kong, Xiaoming Li, and Yan Zhang. 2011. Evolutionary timeline summarization: a balanced optimization framework via iterative substitution. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. ACM, 745–754.
[30] Deyu Zhou, Linsen Guo, and Yulan He. 2018. Neural Storyline Extraction Model for Storyline Generation from News. In *Proceedings of NAACL-HLT*. 1727–1736.
[31] Deyu Zhou, Haiyang Xu, Xin-Yu Dai, and Yulan He. 2016. Unsupervised Storyline Extraction from News Articles.. In *IJCAI*. 3014–3021.

A REPRODUCTION

We present the experiment details to reproduce our results.

Datasets and pre-processing. Both *timeline17* and *crisis* are available at http://l3s.de/~gtran/timeline/. We use spaCy [6] to tokenize news articles into sentences. For temporal tagging, we use HeidelTime [7] to detect all temporal expressions in each sentence. If one sentence contain multiple date expressions, we consider all distinct date-sentence pairs in generating dated sentences $\{(date_i, sentence_i)\}$. Besides, each sentence is also paired with the publication date of the article it appears in.

Evaluation. As suggested at the beginning of Section 5.2 in [25], we set the number of selected dates T to the number of dates in each ground-truth timeline, and the number of sentences per day N to the rounded value of the average number of sentences per date from the corresponding ground-truth timeline. In Table 4 and Table 5, we follow existing works and use ROUGE-1.5.5 to get concatenate ROUGE scores, including ROUGE-1, ROUGE-2 and ROUGE-S*, which ignores date selection in the generated summarization and concatenate all daily summaries together. For comparison with TILSE, we use the evaluation library from the authors [8] for time-sensitive ROUGE scores in Table 6. But different from previous papers, for *Timeline17* dataset, TILSE [11] mixed articles of the same topic from different news agencies together and uses filtered sentence corpus for both datasets. Thus, for a fair comparison, we dump their sentence candidate pool through TILSE code and run our daily summarization on the same sentence candidate pool for each timeline. In speed evaluation, we do not consider the temporal tagging in the pre-processing, and only measure the speed of generation on the tagged sentences for both TILSE and WILSON. The wall time is measured on a 24-core machine.

Implementation details of WILSON. For daily summarization, we group dated sentences $\{(date_i, sentence_i)\}$ by the date to obtain the sentence candidates for each date. Since one sentence can have multiple paired dates, it may appear in multiple daily summaries. When utilizing TextRank [13] to generate daily summaries, we use BM25 [18] scores as edge weight. More specifically, when calculating the edge weight of one sentence to other sentences, we treat the source sentence as query and other sentences as documents, and use its BM25 relevance scores as edge weights. BM25 weights are unsymmetrical, so we build a directed graph for each date and then run the PageRank algorithm to select top sentences as daily summaries. For PageRank algorithm in both date selection and daily summarization, we use the implementation of NetworkX [9] with default damping parameter $\alpha = 0.85$. Code is available at https://github.com/wilson-nts/WILSON.

Implementation details of baselines. Among all the baselines, TILSE is the only one with source code available. Therefore, for all the other baselines, we follow the existing works [9, 25, 27], adopt the conventional experiment setting and directly report the results from previous papers. For the news timeline outputs of TILSE [12] (both TLSCONSTRAINTS and ASMDS), we use the author implementation [10] and their provided configurations [11]. Note that, the TILSE implementation uses the same processing (e.g. caches sentence similarity calculation) to generate multiple timelines that use the same news corpus, therefore, we add the processing time back in measuring the generation time per timeline.

[6] https://spacy.io
[7] https://github.com/HeidelTime/heideltime
[8] https://github.com/smartschat/tilse/tree/master/tilse/evaluation
[9] https://networkx.github.io/
[10] https://github.com/smartschat/tilse
[11] https://github.com/smartschat/tilse/tree/master/configs

Conversational OLAP in Action

Matteo Francia
DISI — University of Bologna, Italy
m.francia@unibo.it

Enrico Gallinucci
DISI — University of Bologna, Italy
enrico.gallinucci@unibo.it

Matteo Golfarelli
DISI — University of Bologna, Italy
matteo.golfarelli@unibo.it

ABSTRACT

The democratization of data access and the adoption of OLAP in scenarios requiring hand-free interfaces push towards the creation of smart OLAP interfaces. In this demonstration we present COOL, a tool supporting natural language *COnversational OLap* sessions. COOL interprets and translates a natural language dialogue into an OLAP session that starts with a GPSJ (Generalized Projection, Selection and Join) query. The interpretation relies on a formal grammar and a knowledge base storing metadata from a multidimensional cube. COOL is portable, robust, and requires minimal user intervention. It adopts an n-gram based model and a string similarity function to match known entities in the natural language description. In case of incomplete text description, COOL can obtain the correct query either through automatic inference or through interactions with the user to disambiguate the text. The goal of the demonstration is to let the audience evaluate the usability of COOL and its capabilities in assisting query formulation and ambiguity/error resolution.

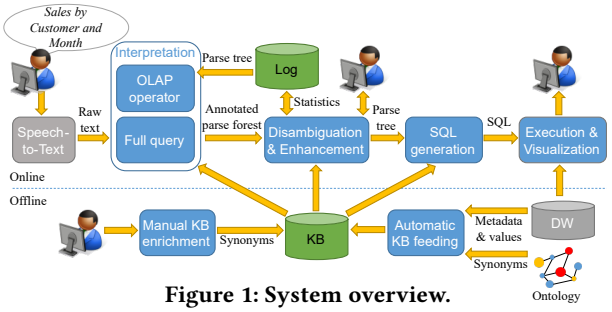


Figure 1: System overview.

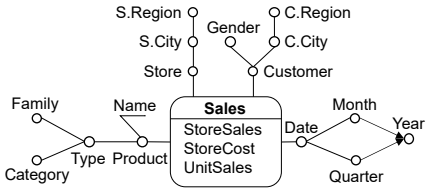


Figure 2: Sales cube in the DW.

1 INTRODUCTION

Following the spreading of analytic tools, a heterogeneous plethora of data scientists is accessing data. However, the gap between data scientists and analytic skills is growing since different types of data require to learn specialized metaphors and formal tools (e.g., SQL language to query relational data). Natural language interfaces are a promising bridge towards the democratization of data access [13]. Rather than demanding vertical skills in computer science and data architectures, natural language is a native "tool" to organize and provide meaningful questions/answers. Interfacing natural language processing (either written or spoken) to database systems opens to new opportunities for data exploration and querying [9]. Actually, in the area of data warehouse, OLAP (On-Line Analytical Processing) is an *"ante litteram"* smart interface, since it supports the users with a "point-and-click" metaphor to avoid writing well-formed SQL queries. Nonetheless, the possibility of having a conversation with a smart assistant to run an OLAP session (i.e., a set of related OLAP queries) opens to new scenarios and applications. It is not just a matter of further reducing the complexity of posing a query: a conversational OLAP system must also provide feedback to refine and correct wrong queries, and it must have memory to relate subsequent requests. A reference application scenario is augmented business intelligence [6], where hand-free interfaces are mandatory.

In this demo paper, we propose COOL to convert natural language into *COnversational OLap* sessions composed of GPSJ queries and analytic operators. GPSJ [8] is the main class of queries used in OLAP since it enables Generalized Projection, Selection and Join operations over a set of tables. Although some natural language interfaces to databases have already been proposed [3], this is the first proposal addressing a full-fledged implementation for OLAP analytical sessions that is:

- *Automated and portable*: by reading metadata (e.g., hierarchy structures, measures, attributes, and aggregation operators) from a ROLAP engine, COOL automatically builds the minimal lexicon involved in the translation.
- *Robust* to user inaccuracies in syntax, OLAP terms, and attribute values, by exploiting metadata and implicit information.
- *Extendable* and easily configurable on a data warehouse (DW) without a heavy manual definition of the lexicon. The minimal lexicon is extendable by importing known ontologies in the Knowledge Base.

COOL's initial proposal in [4] has been now extended by (1) implementing a full-fledged application that supports a complete OLAP session rather than a single query; and (2) providing a visual metaphor based on the Dimensional Fact Model (DFM) [7] to guide user interaction (Figure 2 conceptualizes a multidimensional cube with the DFM formalism). Noticeably, the user interface's effectiveness has been assessed with 40 users, including data scientists and master students with varying skill levels.

The goal of the demonstration is to let the audience evaluate the usability of COOL and its capabilities in assisting query formulation and ambiguity/error resolution. The system is publicly available at https://big.csr.unibo.it/cool.

2 SYSTEM OVERVIEW

Figure 1 sketches a functional view of the architecture. Given a set of multidimensional cubes (DW), we distinguish between an offline phase (to initialize and configure the system) and an online phase (to enable the user interaction). We refer the user to [7] for an explanation of the DW terminology.

2.1 The offline phase

The *offline* phase automatically extracts the set of entities $\mathcal{E}$, i.e., the DW-specific terms used to express the queries. Such information is stored in the knowledge base (KB), which relies on the DFM expressiveness [7]. This phase runs *only* when the DW undergoes modification (e.g., cube schemas or data instances) and extracts all multidimensional metadata. A cube modeled as a *star schema* in a ROLAP engine consists of dimension tables (DTs: the cube hierarchies) and a fact table (FT: the cube). The `Automatic KB feeding` extracts measures (FT columns not included in the primary key), attributes (DT columns), values (distinct instances of the DT columns), and hierarchies (either coded in a specific table or inferred [10]). These elements represent the lexicon necessary to translate natural language into conversational sessions. COOL supports lexicon extension with external synonyms that can be automatically imported from open data ontologies (e.g., Wordnet [11]) to widen the understood language. Besides the domain-specific terminology, the KB also includes the set of standard terms that are domain independent and that do not require any feeding (e.g., group by, where, select). Further enrichment can be optionally carried out manually when the application domain involves a non-standard vocabulary (e.g., the physical names of tables and columns do not match a standard vocabulary).

2.2 The online phase

The *online* phase runs every time a natural language query is issued. In a hand-free scenario (e.g., [5]), the spoken query is initially translated to text by the `Speech-to-text` software module. Since this task is out of our research scope, we exploited the public Web Speech API (https://wicg.github.io/speech-api/). The uninterpreted text is then analyzed by the `Interpretation` step, refined in the `Disambiguation & Enhancement` step, translated by the `SQL generation` step, and finally executed and visualized by the `Execution & Visualization` step.

2.2.1 Interpretation. `Interpretation` consists of two alternative steps. `Full query` interprets the texts describing full queries (which happens when an analytic session starts). `OLAP operator` modifies the latest query when the user states an OLAP operator along a session (i.e., roll-up, drill-down, and slice&dice). The switch between the two steps to manage the conversation (i.e., a dialog between the user and COOL) is modeled by two states: *engage* and *navigate*.

- *Engage*: this is the initial state, in which the system expects a full query to be issued and whose interpretation is demanded to `Full query`. When COOL achieves a successful interpretation (i.e., it is able to run the query) it switches to the *navigate* state.
- *Navigate*: the dialogue evolves by iteratively applying OLAP operators that refine the query (e.g., by aggregating by different levels or narrowing the query selectivity). The management of these steps is demanded to `OLAP operator` until the user resets the session, making COOL return to the *engage* state.

`Full query` and `OLAP operator` follow these steps: (i) `Tokenization & Mapping`, (ii) `Parsing`, and (iii) `Checking & Annotation`.

Tokenization & Mapping. A raw text T is a sequence of single words $T = \langle t_1, ..., t_z \rangle$. The goal of this step is to identify the entities in T, i.e., the only elements that will be involved in the `Parsing` step. Turning a text into a sequence of entities means finding a *mapping* between words in T and $\mathcal{E}$.

Definition 2.1 (Mapping & Mapping function). A mapping function $M(T)$ is a partial function that associates sub-sequences (or *n-grams*)[1] from T to entities in $\mathcal{E}$ such that:

- sub-sequences of T have length n at most;
- the mapping function determines a partitioning of T;
- a sub-sequence $T' = \langle t_i, ..., t_l \rangle \in T$ (with $|T'| \leq n$) is associated to an entity E if and only if $Sim(T', E) > \alpha$ (where $Sim()$ is a similarity function, later defined) and $E \in TopN(\mathcal{E}, T')$ (where $TopN(\mathcal{E}, T')$ is the set of N entities in $\mathcal{E}$ that are the most similar to T' according to $Sim(T', E)$).

The output of a mapping function is a sequence $M = \langle E_1, ..., E_l \rangle$ on $\mathcal{E}$ that we call a *mapping*.

The similarity function $Sim()$ is based on the Levenshtein distance and keeps token permutation into account to make similarity robust to token permutations (e.g., sub-sequences $\langle P., Edgar \rangle$ and $\langle Edgar, Allan, Poe \rangle$ must result similar).

Several mappings might exist between T and $\mathcal{E}$ since Definition 2.1 admits sub-sequences of variable lengths (corresponding to different partitionings of T) and associates the top similar entities to each sub-sequence. This increases the interpretation robustness since COOL chooses the best mapping through a scoring function. Given a mapping $M = \langle E_1, ..., E_m \rangle$, its score $Score(M)$ (i.e., the sum of entity similarities) is higher when M includes several entities with high similarity values. Intuitively, the higher the mapping score, the higher the probability to determine an optimal interpretation.

Parsing. Parsing validates the syntactical structure of a mapping against a formal grammar and outputs a data structure called *parse tree* that is later used to translate a mapping into SQL.

In `Full query`, `Parsing` is responsible for the interpretation of a complete GPSJ query stated in natural language. A GPSJ query contains a measure clause (MC) and optional group-by (GC) and selection (SC) clauses. Parsing a full query means searching in a mapping the complex syntax structures (i.e., clauses) that build-up the query. Given a mapping M, the output of the parser is a parse tree PT_M, i.e., an ordered tree that represents the syntactic structure of a mapping according to the grammar from Figure 3. To the aim of parsing, entities are terminal elements in the grammar.

In `OLAP operator`, `Parsing` is responsible for searching the syntactic structures of the OLAP operators that build-up the conversation. The grammar is described in Figure 4. Our conversation steps are inspired to well-known OLAP visual interfaces (e.g., Tableau[2]). To apply an OLAP operator, COOL must be in the state *navigate*, i.e., a full GPSJ query has been already successfully interpreted and translated into a parse tree PT_C that acts as a context for the operator. The output of the parser is a parse tree PT_M that is used to update PT_C (see Section 2.2.3).

Both GPSJ grammars are LL(1)[3] [2], not ambiguous (i.e., each mapping admits, at most, a single parse tree PT_M), and can be parsed by an LL(1) parser with linear complexity [2]. If the input mapping M is fully parsed, PT_M includes all the entities as leaves. Conversely, if only a portion of the input belongs to the grammar, an LL(1) parser produces a partial parsing, meaning that it returns a parse tree including the portion of the input mapping

[1] The term *n*-gram is used as a synonym of sub-sequence in the area of text mining.
[2] https://www.tableau.com/
[3] The rules presented in Figure 3 do not satisfy LL(1) constraints for readability reasons. It is easy to turn such rules in an LL(1) complaint version, but the resulting rules are much more complex to be read and understood.

⟨GPSJ⟩ ::= ⟨MC⟩⟨GC⟩⟨SC⟩ | ⟨MC⟩⟨SC⟩⟨GC⟩ | ⟨SC⟩⟨GC⟩⟨MC⟩ | ⟨SC⟩⟨MC⟩⟨GC⟩
| ⟨GC⟩⟨SC⟩⟨MC⟩ | ⟨GC⟩⟨MC⟩⟨SC⟩ | ⟨MC⟩⟨SC⟩ | ⟨MC⟩⟨GC⟩
| ⟨SC⟩⟨MC⟩ | ⟨GC⟩⟨MC⟩ | ⟨MC⟩

⟨MC⟩ ::= (⟨Agg⟩⟨Mea⟩ | ⟨Mea⟩⟨Agg⟩ | ⟨Mea⟩ | ⟨Cnt⟩⟨Fct⟩ | ⟨Fct⟩⟨Cnt⟩
| ⟨Cnt⟩⟨Attr⟩ | ⟨Attr⟩⟨Cnt⟩)+

⟨GC⟩ ::= "*group by*" ⟨Attr⟩+

⟨SC⟩ ::= "*where*" ⟨SCA⟩

⟨SCA⟩ ::= ⟨SCN⟩ "*and*" ⟨SCA⟩ | ⟨SCN⟩

⟨SCN⟩ ::= "*not*" ⟨SSC⟩ | ⟨SSC⟩

⟨SSC⟩ ::= ⟨Attr⟩⟨Cop⟩⟨Val⟩ | ⟨Attr⟩⟨Val⟩ | ⟨Val⟩⟨Cop⟩⟨Attr⟩ | ⟨Val⟩⟨Attr⟩ | ⟨Val⟩

⟨Cop⟩ ::= " = " | " <> " | " > " | " < " | " ≥ " | " ≤ "

⟨Agg⟩ ::= "*sum*" | "*avg*" | "*min*" | "*max*" | "*stdev*"

⟨Cnt⟩ ::= "*count*" | "*count distinct*"

⟨Fct⟩ ::= *Domain-specific facts*

⟨Mea⟩ ::= *Domain-specific measures*

⟨Attr⟩ ::= *Domain-specific attributes*

⟨Val⟩ ::= *Domain-specific values*

Figure 3: Backus-Naur representation of the `Full query` grammar. Entities from the `KB` are terminal symbols.

⟨OPERATOR⟩ ::= ⟨DRILL⟩ | ⟨ROLLUP⟩ | ⟨SAD⟩ | ⟨ADD⟩ | ⟨DROP⟩ | ⟨REPLACE⟩

⟨DRILL⟩ ::= "*drill*" ⟨Attr⟩$_{from}$ "*to*" ⟨Attr⟩$_{to}$ | "*drill*" ⟨Attr⟩

⟨ROLLUP⟩ ::= "*rollup*" ⟨Attr⟩$_{from}$ "*to*" ⟨Attr⟩$_{to}$ | "*rollup*" ⟨Attr⟩

⟨SAD⟩ ::= "*slice*" ⟨SSC⟩

⟨ADD⟩ ::= "*add*" (⟨MC⟩ | ⟨Attr⟩ | ⟨SSC⟩)

⟨DROP⟩ ::= "*drop*" (⟨MC⟩ | ⟨Attr⟩ | ⟨SSC⟩)

⟨REPLACE⟩ ::= "*replace*" (⟨MC⟩$_{old}$ "*with*" ⟨MC⟩$_{new}$ | ⟨Attr⟩$_{old}$ "*with*" ⟨Attr⟩$_{new}$
| ⟨SSC⟩$_{old}$ "*with*" ⟨SSC⟩$_{new}$)

Figure 4: Backus-Naur representation of the `OLAP operator` grammar. We omit ⟨MC⟩, ⟨Attr⟩, ⟨SSC⟩ in common with Figure 3.

that belongs to the grammar (i.e., the *PT* rooted in ⟨GPSJ⟩). The remaining entities can be either singleton or complex clauses that could not be connected to the main parse tree. We will call *parse forest* PF_M the union of the parse tree with residual clauses. Obviously, if all the entities are parsed, it is $PF_M = PT_M$. Considering the whole forest rather than the simple parse tree enables disambiguation and errors to be recovered in the `Disambiguation & Enhancement` step. Henceforth, we refer to the parser's output as a parse forest independently of the presence of residual clauses.

2.2.2 Disambiguation & enhancement. Due to natural language ambiguities, speech-to-text inaccuracies, and wrong query formulations, parts of the text can be misunderstood. The reasons behind the misunderstandings are manifold, including (but not limited to) a wrong usage of aggregation operators (e.g., summing non-additive measures), inconsistencies between attributes and values in selection predicates (e.g., filtering on product *"New York"*), or grouping by a descriptive attribute. Such parts of the parse forest are annotated as ambiguities. The `Disambiguation & Enhancement` step solves ambiguities (if any) automatically whenever possible (by exploiting implicit information) or by asking appropriate questions to the user. Through disambiguation, the parse forest PF_M is reduced to a single parse tree PT_M.

2.2.3 SQL generation. `SQL generation` translates a full-query parse tree into an executable SQL query. If an OLAP operator has been submitted, the context parse tree PT_C must be updated according to the OLAP operator parse tree PT_M. All the OLAP operators can be implemented atop the addition/removal of new/existing nodes in PT_C. We apply a depth-first search algorithm to retrieve the clauses interested by the OLAP operator. We recall that adding a new clause to ⟨GPSJ⟩ (e.g., "add city" requires to add the attribute City) requires to append the new clause to the existing ⟨MC⟩/⟨GC⟩/⟨SC⟩ (and to create it if it does not exist in ⟨GPSJ⟩). Given a full query parse tree PT_M, the generation of its corresponding SQL requires to fill in the `SELECT`, `WHERE`, `GROUP BY` and `FROM` statements. The SQL generation applies to both star and snowflake schemas [7] and is done as follows:

- `SELECT`: measures and aggregation operators from ⟨MC⟩ and attributes in the group by clause ⟨GC⟩;
- `WHERE`: predicates from the selection clause ⟨SC⟩;
- `GROUP BY`: attributes from the group by clause ⟨GC⟩;
- `FROM`: measures, attributes, and values identify fact and dimension tables. The join path is identified by following the referential integrity constraints.

3 VISUALIZATION METAPHOR

The obtained query is run on the `DW` and the results are reported to the user by the `Execution & Visualization` software module. The visual interaction relies on the DFM [7] (Figures 5 and 6), which natively provides a graphical representation for multidimensional cubes and queries: such representation is conceptual and user-oriented, and its effectiveness is confirmed by its adoption in commercial tools (e.g., https://www.indyco.com/) for both modeling and descriptive purposes. With reference to Figure 2, the DFM explicitly represents the cube as a rectangle (i.e., Sales) including the measure names (e.g., StoreSales) surrounded by hierarchies organized in many-to-one acyclic graphs (e.g., Products are grouped in Types). This representation includes all the elements necessary to formulate a GPSJ query, namely measure clauses, group-by clauses, and selection clauses on both measures and dimensional attributes.

At first, users are asked to submit the natural language description of a full query (e.g., *"return the medium costs for Beer and Wine by gender"*). Once COOL interprets the natural language query, it shows in green the entities that are fully-understood (`product_category=Beer and Wine`, `store_cost(avg)`, and `gender` in Figure 5) and, if no ambiguities exist, it also returns the query result. If some ambiguities exist, COOL shows in yellow the ambiguous elements on the DFM one by one. For instance, when users ask for *"average costs in the USA"* (Figure 6), COOL shows that `store_cost(avg)` is correctly understood, while `USA` is ambiguous since it belongs to both attributes `store_country` and `country`. In the case of ambiguities, besides color codes, COOL notifies the user of the encountered ambiguity and enables multiple resolutions (e.g., either picking the correct attribute or dropping the clause). COOL also shows the parse tree on request, enabling more skilled users to understand how natural language is interpreted. After issuing a full query, the conversational session proceeds by describing further analytic steps (e.g., *"generalize product subcategory to category"* or more briefly *"generalize product subcategory"*).

We tested COOL against the Foodmart [1] cube with 40 users, mainly master students in data science, with basic or advanced knowledge of business intelligence and data warehousing. On a scale from 1 (very poor) to 5 (very high), on average, users scored 3.60 ± 0.7 their familiarity with the English language, and 3.28 ± 1.1 their familiarity with the OLAP paradigm. Users evaluated the responsiveness and user-friendliness of COOL as 4.09 ± 0.85, and the overall user experience (e.g., the perceived translation accuracy) as 3.82 ± 0.91, confirming a good — or even

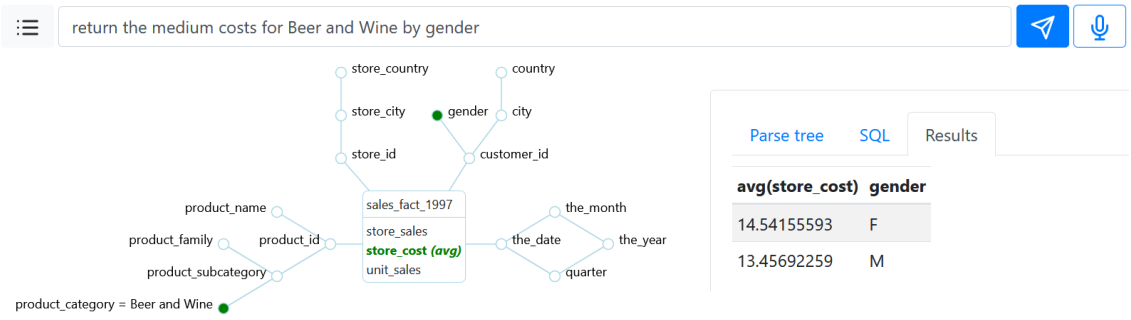


Figure 5: COOL returns the results for a fully-interpreted query.

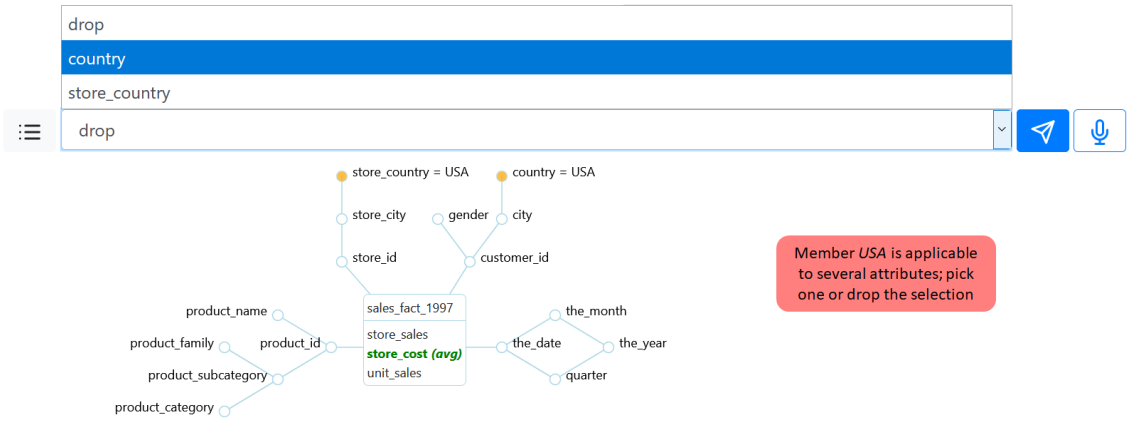


Figure 6: COOL shows a hint to disambiguate the *member* "USA".

optimal — experience. Finally, the average response time is less than 1 second for mappings including up to 9 entities.

4 DEMO PROPOSAL

In the demonstration, we develop an experience to showcase the translation of natural language dialogues into analytical sessions. In particular, we consider the usability criteria of a visual interface from [12]. Usability is assessed by *learnability* (how easily novel users accomplish basic tasks), *efficiency* (how quickly users can perform tasks), *error recovery* (how many errors users make and how easily can they recover from these errors), and *satisfaction* (how pleasant it is to use the interface). Two distinct scenarios are proposed to assess these functionalities.

In the *guided* scenario, we drive the formulation of analytic sessions on the Sales cube. In particular, we provide users with three formal definitions of GPSJ queries in the format $GPSJ = \{\{GC\}, \{SC\}, \{MC\}\}$; users interact with the system by formulating such queries in natural language. Once the query description is submitted (spoken or written), COOL drives users in the resolutions of ambiguities, if any, through a question-answer approach. Then, further analytics steps are shown to the user, who is required to abstract and describe the operators necessary to accomplish such steps. In such a way, we address *learnability* and *efficiency*, allowing users to approach COOL in a user-friendly manner.

In the following *unguided* scenario, users are to freely interact with COOL starting only with a generic analytic task (e.g., *"Investigate sales drop. This might depend on examining products sales over time"*). This requires to formulate custom analytic goals and sessions toward such goals (e.g., sales drop for the category "Beer and Wine" might be caused by a drop in "Wine" sales). As we expect users to encounter ambiguities while freely navigating the cube (e.g., in case of homonyms and synonyms), we address the robustness of COOL through its *error recovery* capability. Also, we assess user *satisfaction* by understanding how easy it is for the user to accomplish their analytic goal (i.e., how many steps it takes).

REFERENCES

[1] [n.d.]. Foodmart. https://github.com/julianhyde/foodmart-data-mysql. Accessed: 18/01/2021.

[2] John C. Beatty. 1982. On the relationship between LL(1) and LR(1) grammars. *J. ACM* 29, 4 (1982), 1007–1022.

[3] Kedar Dhamdhere, Kevin S. McCurley, Ralfi Nahmias, Mukund Sundararajan, and Qiqi Yan. 2017. Analyza: Exploring Data with Conversation. In *Proc. IUI*. ACM, New York, NY, USA, 493–504.

[4] Matteo Francia, Enrico Gallinucci, and Matteo Golfarelli. 2020. Towards Conversational OLAP. In *Proc. DOLAP@EDBT/ICDT (CEUR Workshop Proceedings)*, Vol. 2572. CEUR-WS.org, Copenhagen, Denmark, 6–15.

[5] Matteo Francia, Matteo Golfarelli, and Stefano Rizzi. 2019. Augmented Business Intelligence. In *Proc. DOLAP@EDBT/ICDT (CEUR Workshop Proceedings)*, Vol. 2324. CEUR-WS.org, Lisbon, Portugal, 1–10.

[6] Matteo Francia, Matteo Golfarelli, and Stefano Rizzi. 2020. A-BI$^+$: A framework for Augmented Business Intelligence. *Inf. Syst.* 92 (2020), 101520.

[7] Matteo Golfarelli, Dario Maio, and Stefano Rizzi. 1998. The Dimensional Fact Model: A Conceptual Model for Data Warehouses. *Int. J. Cooperative Inf. Syst.* 7, 2-3 (1998), 215–247.

[8] Ashish Gupta, Venky Harinarayan, and Dallan Quass. 1995. Aggregate-Query Processing in Data Warehousing Environments. In *Proc. VLDB*. Morgan Kaufmann, San Francisco, CA, USA, 358–369.

[9] Fei Li and H. V. Jagadish. 2016. Understanding Natural Language Queries over Relational Databases. *SIGMOD Record* 45, 1 (2016), 6–13.

[10] Heikki Mannila and Kari-Jouko Räihä. 1994. Algorithms for Inferring Functional Dependencies from Relations. *Data Knowl. Eng.* 12, 1 (1994), 83–99.

[11] George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[12] Jakob Nielsen. 1993. *Usability engineering*. Academic Press.

[13] Yu Su. 2018. *Towards Democratizing Data Science with Natural Language Interfaces*. Ph.D. Dissertation. UC Santa Barbara.

Smart City Data Analysis via Visualization of Correlated Attribute Patterns

Yuya Sasaki[1*], Keizo Hori[1*], Daiki Nishihara[1*], Sora Ohashi[1*], Yusuke Wakuta[1*], Kei Harada[1], Makoto Onizuka[1], Yuki Arase[1], Shinji Shimojo[2], Kenji Doi[3], He Hongdi[4], Zhong-Ren Peng[5]
[1]Graduate School of Information Science and Technology, Osaka University, Suita, Japan, [2]Cyber media center, Osaka University, Suita, Japan, [3]Graduate School of Engineering, Osaka University, Suita, Japan, [4]Center for Intelligent Transportation Systems and Unmanned Aerial Systems Applications Research, Shianghai Jiao Tong University, Shanghai, China, and [5]International Center for Adaptation Planning and Design, University of Florida, Gainesville, USA.
{sasaki, hori.keiso, nishihara.daiki, ohashi.sora, wakuta.yusuke, harada.kei, onizuka, arase}@ist.osaka-u.ac.jp, shimojo@cmc.osaka-u.ac.jp, doi@civil.eng.osaka-u.ac.jp, hongdihe@sjtu.edu.cn, zpeng@ufl.edu

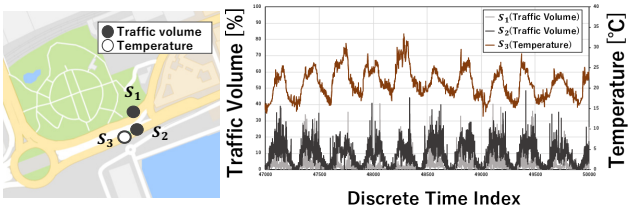
ABSTRACT

Urban conditions are monitored by a wide variety of sensors that measure several attributes, such as temperature and traffic volume. The correlations of sensors help to analyze and understand the urban conditions accurately. The correlated attribute pattern (CAP) mining discovers correlations among multiple attributes from the sets of sensors spatially close to each other and temporally correlated in their measurements. In this paper, we develop a visualization system for CAP mining and demonstrate analysis of smart city data. Our visualization system supports an intuitive understanding of mining results via sensor locations on maps and temporal changes of their measurements. In our demonstration scenarios, we provide four smart city datasets collected from China and Santander, Spain. We demonstrate that our system helps interactive analysis of smart city data.

1 INTRODUCTION

Many cities have started smart city initiatives and installed a wide variety of sensors that measure several attributes, such as traffic volume and temperature. The collected data from smart cities is used for continuously and cooperatively monitoring urban conditions, such as the distribution of air pollution, the transition of traffic volume, and the change of citizen activity. Researchers and municipalities analyze smart city data and make a decision for urban planning. For example, environmental researchers in Shanghai Jiao Tong university analyze the relationships between traffic and air pollution [5, 7]. Santander, Spain monitors the traffic volumes within the city and informs people of the real-time traffic information [6]. They work on obtaining useful patterns in cities by using database and data mining techniques.

Smart city data has spatial and temporal information. For analysing spatio-temporal data, we proposed *correlated attribute pattern (CAP) mining* [2, 3]. CAP mining aims to find correlated attributes of sensors that are spatially close to each other and whose measurements temporally co-evolve. We developed an efficient algorithm for CAP mining, called Miscela and presented that the correlated attribute patterns can discover useful knowledge from smart city data. We show an example that illustrates the effectiveness of CAP mining.



(a) Sensor location (b) Correlation

Figure 1: The correlation between traffic volume and temperature in Santander [2]

Example 1.1. Figure 1 shows locations of three sensors s_1, s_2, and s_3 in Santander and these measured values. s_1 and s_2 measure traffic volume and s_3 measures temperature. These sensors are spatially close to each other, and the measurements of them co-evolve frequently (i.e., change the values simultaneously). The CAP mining can discover correlated patterns among traffic volume and temperature measured by the three sensors. Municipalities can understand that traffic behavior in the area is correlated to temperature from the CAP.

Contribution: In this paper, we develop a visualization system for CAP mining, called Miscela-V, to support an intuitive analysis of smart city data. Miscela-V has the following characteristics:

- Miscela-V natively supports CAP mining with user-specified parameters.
- Miscela-V visualizes sensor locations on a map and temporal changes of sensor measurements.
- Miscela-V caches results of CAP mining and reuses the cached results for efficient interactive analysis.

Our system supports intuitive understanding of analytic results via visualization. We demonstrate an analysis of smart city data by using our system. We use two different scale datasets: Santander (i.e., city size) and China (i.e., country size). Our system is effective for any space and time scales such as daily city-scale and minutely country-scale datasets. For further investigation, we open our source codes[1].

This work is a collaborated work with researchers in database, environmental, and urban science fields, so we validated that Miscela-V is effective for environmental and urban science studies. Through the demonstration of Miscela-V, we expect that Miscela-V helps researchers in more other fields for accelerating their analysis.

[1]https://github.com/OnizukaLab/MISCELA-v

* These authors contributed equally. Yuya Sasaki is the corresponding author.

Related systems: There are several systems for visualizing spatio-temporal data (e.g., [1, 4, 8]). Some systems support spatial-temporal pattern mining but no systems support CAP mining. The novelty of our system is that it focuses on CAP mining with efficient interactive analysis.

Organization: The rest of this paper is organized as follows. We explain the CAP mining and the CAP mining method Miscela in Section 2 as preliminaries. Then, we present a visualization system Miscela-V in Section 3. After that, we show our demonstration plan in Section 4, followed by the conclusion in Section 5.

2 PRELIMINARIES

We explain CAP mining and Miscela as preliminaries.

2.1 CAP mining

We consider a sensor set in a geographical region. Each sensor has longitude and latitude as spatial information. It measures a specific attribute, such as temperature, traffic volume, and PM2.5. Each sensor is synchronized, that is, it measures its sensor value at a certain interval. We define that measurements are co-evolved if they increase/decrease at the same timestamp.

The CAP mining aims for discovering spatially and temporally correlated environmental properties such that multiple sensors measure those attributes that satisfy the following conditions: (1) the set of sensors are located at spatially close locations to each other, (2) the measurements of the sensors co-evolve frequently, and (3) the set of attributes measured by the sensors includes multiple attributes. The CAP mining restricts the correlation between different attributes to support diversified analysis of smart cities. This restriction can be easily removed.

CAP mining has several parameters for obtaining CAPs that users want. We here summarize parameters and their impacts on the number of CAPs to be discovered.

- Evolving rate ε: The CAP mining removes slight changes of measurements by specifying ε. If the amount of changes from the previous timestamp is smaller than ε, the timestamps are evaluated as that the measurements do not change. If ε is large, sensors likely co-evolve, so the number of CAPs likely becomes large.
- Distance threshold η: η gives a criterion of close sensors. If a distance between the two sensors is less than η, we define that they are close. If η is large, many sensors are spatially close to each other.
- The maximum number of CAP attributes μ: μ restricts the number of attributes in CAPs. The CAP mining discovers correlations among not larger than μ attributes.
- The minimum support ψ: ψ is the minimum support. If measurements of two sensors co-evolve at more than ψ timestamps, they are co-evolving sensors. If ψ is small, many sensors become co-evolving sensors, and thus the number of CAPs likely becomes large.

Since the sensitivity of parameters depends on datasets, it is necessary to support interactive analysis. Please see more detailed definitions in [2].

2.2 MISCELA: an efficient algorithm for CAP mining

Miscela supports efficient computation for CAP mining, which comprises the following four steps.

(1) **Linear segmentation**: We filter uninteresting data fluctuation by applying a linear segmentation algorithm to time series data.
(2) **Extracting evolving timestamps**: We extract evolving timestamps in the measurements of all sensors by using the given evolving rate ε.
(3) **Discovering spatially connected sets of sensors**: Since CAPs are discovered only from spatially connected sets, we divide a given sensor set into spatially close sensors to restrict the search space.
(4) **CAP search**: For each set of spatially close sensors, we search for CAPs. We recursively conduct the CAP search with gradually expanding spatially close sensors according to a tree structure for CAP mining.

Please see more detailed and precise procedures in [2].

3 MISCELA-V: VISUALIZING SYSTEM

We present our visualization system, which we call Miscela-V. The purposes of Miscela-V is (1) to easily find CAPs in users' datasets, (2) to visually understand the CAPs, and (3) to efficiently support interactive analysis. First, Miscela-V natively supports CAP mining. It visualizes locations of sensors and changes of their measurements to understands reasons why these attributes are correlated. In addition, since Miscela may take a large execution depending on their parameters, it has a caching mechanism for efficient interactive CAP mining.

3.1 System overview

Figure 2 shows an overview of Miscela-V. Miscela-V has three main processes to visualize CAP mining results. First, we upload datasets to the system. Then, we input parameters of CAP mining for obtaining appropriate results. Finally, we can see the CAP results on a map and the temporal behaviors of their measurements. Since our system supports interactive analysis, data and CAPs are stored in databases. Users can easily change parameters to check CAPs in different parameters. If users specify the parameters of CAPs stored in databases, we can immediately see CAPs without processing Miscela.

Figure 3 shows a visualization of sensor locations and temporal measurements. Figures (A) and (B) show sensor locations, and three sensors are highlighted. When we click a sensor in the map, sensors are highlighted if their measurements are correlated to measurements of the clicked sensor. In addition, we can see the attributes of correlated sensors. Figures (C) and (D) show temporal behaviors of measurements, which we can zoom in and zoom out. In (D), you can see that three measurements frequently increase/decrease together. Our visualization helps to intuitively understand correlations among sensors.

3.2 Data upload

We can easily upload our datasets via a user interface that provides two ways of data upload: drag-and-drop and selecting files from finder. For uploading datasets, we need to prepare three files; data.csv, location.csv, and attribute.csv. data.csv lists the set of measurements at all timestamps. We note that timestamps must be the same time intervals, and sensor values are null if the sensors do not have the sensor values at timestamps. location.csv lists the sensor information; identifier, attribute, and location. attribute.csv lists all attributes in the datasets. Each file should have the following formats:

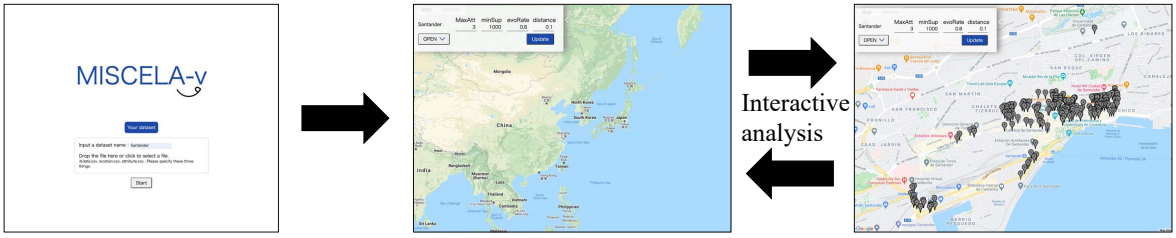


Figure 2: An overview of MISCELA-V

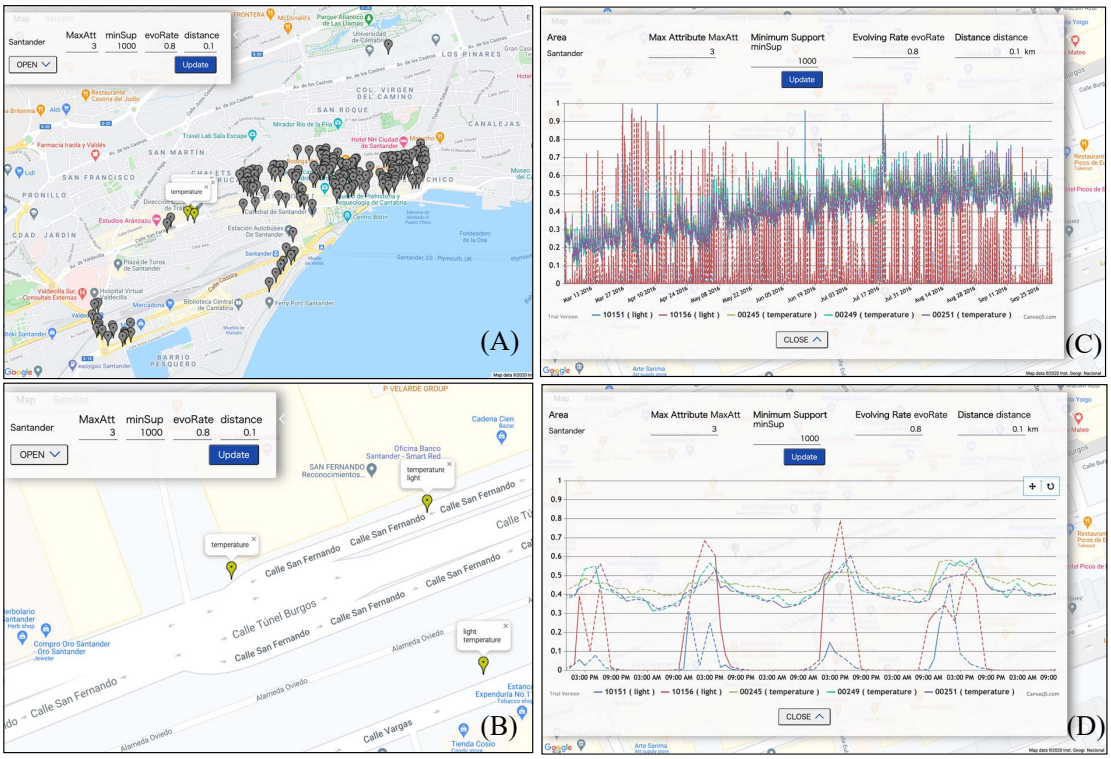


Figure 3: Visualization of CAP mining results

data.csv

```
id,attribute,time,data
00000,temperature,2016-03-01 00:00:00,null
00000,temperature,2016-03-01 01:00:00,9.87
...
```

location.csv

```
id,attribute,lat,lon
00000,temperature,43.46192,-3.80176
00001,temperature,43.46212,-3.79979
...
```

attribute.csv

```
temperature
light
...
```

The data.csv might be very large. For scalably uploading large datasets, we divide the file into 10,000 lines and send each divided set to our system. Each dataset is stored in databases, and thus we can use the dataset without re-uploading by specifying the dataset name.

3.3 Caching mechanism

MISCELA may take a long time for finding CAPs depending on data and user-specified parameters. For efficient interactive analysis, MISCELA-V caches CAP mining results and reuses the cached results if users specify the same parameter setting. This caching mechanism accelerates the analytic process and reduces the computational costs when the front end receives multiple requests at the same time.

We store the name of the dataset, parameters, and CAPs (i.e., a set of sets of sensors) to the database. Before computing CAPs by MISCELA, our system searches for CAPs with the same parameters and the name of the dataset from the database. Since interactive analysis could input the same parameters to compare results repeatedly, the caching mechanism supports more efficient data analysis.

3.4 Implementation

We use MongoDB as database management systems and django as API servers. Miscela is implemented by Python, and a map visualization is implemented by JavaScript, jQuery, and Google Map API. Miscela returns a set of sets of sensors as CAPs that might include many sensors (or empty), and its format is JSON. Since RDBMS is not suitable for Miscela outputs, we select MongoDB to store datasets and CAP results. Since we design that these components are connected by APIs, we can modify each component individually.

4 DEMONSTRATION PLAN

For Miscela-V demonstrations, we use smart city data in Santander and China, as a case study. We will introduce the system architecture, the analytic process, and how to use our system to find knowledge. Attendees can interact with our system to perform analysis using the data. For example, since Miscela-V can show temporal changes of sensors' measurements, we can analyze the difference of measurements before/after COVID-19. The attendees will interactively discover CAPs of smart city data.

The attendees can use the following datasets[2]:

- **Santander** includes 552 sensors in Stantander, Spain from 2016 March 1st to September 30th. The number of records is 2,329,936. Attributes are temperature, light, sound, traffic volume, and humidity.
- **China6** includes 9,438 sensors in China from 2016 September 1st to 2018 October 31st. The number of records is 6,889,740. Attributes are PM2.5, SO2, NO2, CO, and O3.
- **China13** includes 4,810 sensors. The period is the same as China6. The number of records is 3,511,300. Attributes are additionally included in temperature, humidity, air pressure, daylight, rainfall percentage, rain volume, and wind speed.
- **COVID-19** includes 12 sensors in Shanghai and Guangzhou, China from 2020 January 1st to June 30th. The number of records is 52,261. Attributes are PM2.5, PM10, SO2, NO2, CO, and O3. This data includes the period after and before spreading COVID-19.

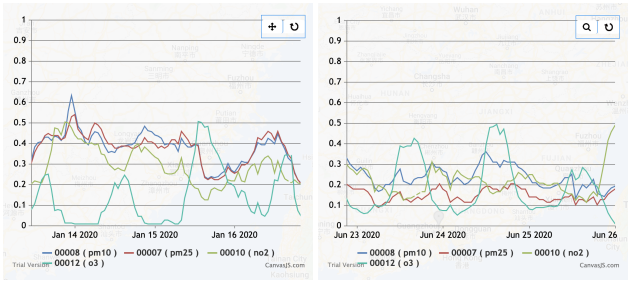
We plan to demonstrate the following case studies.

Interactive analysis: In this demonstration, we first provide interactive analysis to upload datasets, input parameters, and view CAP results. Attendees can freely use our system and try to find interesting patterns in our datasets. First, attendees set the parameters for finding CAPs and see the visualization of the results. Second, the attendees can investigate why the CAPs are discovered by visualizing the temporal behavior of measurements of sensors. Since our system highlights sensors that are correlated, they can understand what sensors are correlated intuitively.

Santander dataset: a single city data analysis: This scenario aims to find interesting knowledge within Santander. Attendees will find interesting CAPs from Santander datasets and investigate the results via visualization. For example, we can find correlated patterns among temperatures and traffic volumes and among light and temperature.

China dataset: multiple cities data analysis: This scenario aims to find interesting knowledge among many cities in China. In particular, attendees can intuitively understand that two sensors are correlated even if they are distant from each other. Furthermore, sensors are not correlated if two sensors are vertically (north and south) close to each other, but if sensors are horizontally (east and west) close, they are correlated. These are often caused by wind directions. We can understand that wind directions affect to air quality from the CAPs. Our system supports for understanding reasons why sensors are correlated and not correlated.

[2]We consider sensors with different attributes as different sensors even if they are located at the same location.



(a) Before (b) After

Figure 4: An example of correlation pattern changes before/after spreading COVID19

COVID-19 analysis: COVID-19 dataset includes the period before and after spreading COVID-19. Attendees can know that levels of air pollution change due to spreading COVID-19. Figure 4 shows the correlation patterns before and after COVID-19. From these results, we can visually understand that our activity changes affect not only the amounts of air pollutants but also their correlation patterns.

5 CONCLUSION

In this paper, we introduced a visualization system Miscela-V for CAP mining and demonstrated the data analysis of smart city via Miscela-V. We plan to continuously extend our system to improve usability and add additional data mining techniques, based on user feedback. We hope that our system accelerates data analysis in many research fields.

Acknowledgements This work was supported by JSPS KAKENHI Grant Numbers JP20H00584.

REFERENCES

[1] Natalia Andrienko and Gennady Andrienko. 2013. A visual analytics framework for spatio-temporal analysis and modelling. *Data Mining and Knowledge Discovery* 27, 1 (2013), 55–83.
[2] Kei Harada, Yuya Sasaki, and Makoto Onizuka. 2019. MISCELA: Discovering correlated attribute patterns in time series sensor data. In *MDM*. 72–80.
[3] Kei Harada, Yuya Sasaki, and Makoto Onizuka. 2020. MISCELA: discovering simultaneous and time-delayed correlated attribute patterns. *Distributed and Parallel Databases* (2020), 1–28.
[4] Tomislav Hengl, Pierre Roudier, Dylan Beaudette, Edzer Pebesma, et al. 2015. plotKML: Scientific visualization of spatio-temporal data. *Journal of Statistical Software* 63, 5 (2015), 1–25.
[5] Baoguo Jiang, Song Liang, Zhong-Ren Peng, Haozhe Cong, Morgan Levy, Qu Cheng, Tianbing Wang, and Justin V Remais. 2017. Transport and public health in China: the road to a healthy future. *The Lancet* 390, 10104 (2017), 1781–1791.
[6] Luis Sanchez, Luis Muñoz, Jose Antonio Galache, Pablo Sotres, Juan R Santana, Veronica Gutierrez, Rajiv Ramdhany, Alex Gluhak, Srdjan Krco, and Evangelos Theodoridis. 2014. SmartSantander: IoT experimentation over a smart city testbed. *ELSEVIER Computer Networks* 61 (2014), 217–238.
[7] Hong-Wei Wang, Xiao-Bing Li, Dongsheng Wang, Juanhao Zhao, Hong di He, and Zhong-Ren Peng. 2020. Regional prediction of ground-level ozone using a hybrid sequence-to-sequence deep learning approach. *Journal of Cleaner Production* 253, 119841 (2020), 1–12.
[8] Liu Xiufeng, Zhibin Niu, Linda Yang, Junqi Wu, Dawei Cheng, and Xin Wang. 2020. VAP: a visual analysis tool for energy consumption spatio-temporal pattern discovery. In *EDBT*. 579–582.

SciNeM: A Scalable Data Science Tool for Heterogeneous Network Mining

Serafeim Chatzopoulos
Univ. of the Peloponnese &
"Athena" RC
schatzop@uop.gr

Thanasis Vergoulis
"Athena" RC
vergoulis@athenarc.gr

Panagiotis Deligiannis
"Athena" RC
deligianp@athenarc.gr

Dimitrios Skoutas
"Athena" RC
dskoutas@athenarc.gr

Theodore Dalamagas
"Athena" RC
dalamag@athenarc.gr

Christos Tryfonopoulos
Univ. of the Peloponnese
trifon@uop.gr

ABSTRACT

Heterogeneous Information Networks (HINs) provide a natural way to represent various relationships between entities of different types, thus they are valuable in many domains. Extracting knowledge from HINs typically relies on the concept of metapaths, which are paths in the network schema denoting relations of different semantics among entities. Moreover, real-world HINs are often extremely large, containing millions of nodes and edges. Thus, exploring HINs not only requires interdisciplinary expertise, being able both to interpret and select appropriate metapaths in the network, but also to run the analysis in an efficient and scalable manner. Since there is a lack of tools to facilitate this task, we present SciNeM, an open source, publicly available, scalable analysis tool for metapath-based knowledge discovery in HINs.

1 INTRODUCTION

Many modern applications rely on analysing large amounts of data that comprise multiple types of entities and relationships between them. For instance, *data-driven science*, which has become a very popular and effective paradigm for scientific research, is based on computationally exploring large heterogeneous datasets. Also, the foundations of the *Fourth Industrial Revolution* heavily rely on *data science* techniques for data-driven decision making based on large heterogeneous datasets from multiple sources.

Heterogeneous Information Networks (HINs) provide a way to represent such complex information. They are graphs comprising multiple types of nodes and relationships between them [7]. An example HIN is illustrated in Figure 1, representing the interactions of genes (G) with a class of biomolecules called miRNAs (M) and their relationship with particular biological processes (P) and diseases (D)[1]. It contains 4 distinct node types (G, M, P, D) and 3 distinct types of (bidirectional) relationships (GM, GP, GD).

Various data science methods to analyse HINs and facilitate knowledge discovery from them have been proposed [6, 8, 11]. These typically rely on the concept of *metapaths*: paths in the HIN schema that represent types of entity relationships with particular semantics. For instance, two interesting metapaths in the HIN of Figure 1 are GPG and $MGDGM$. The former connects genes based on the processes they are involved in (i.e., strongly connected genes based on it may share common functionalities).

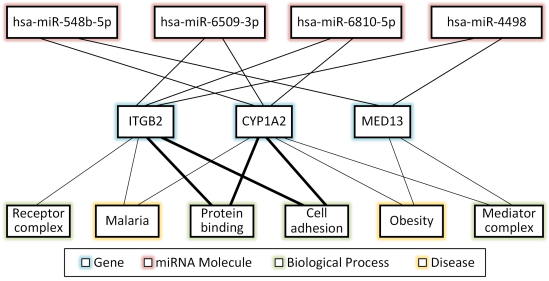


Figure 1: An example HIN.

The latter links miRNAs based on the diseases which relate to the genes with which they interact.

Many HIN analysis algorithms use metapaths as input; the metapath-based connectivity can be used to define a measure for node *similarity search* [8] or *similarity join* [11] or to rank nodes based on their centrality in a metapath-defined network [6]. In the previous example, using the metapath GPG for similarity join could reveal that genes *ITGB2* and *CYP1A2* are similar since they are involved in two common processes. Moreover, to further elaborate the analysis, it is often useful to apply constraints to a given metapath (e.g., in the previous example, to consider only metapath instances involving Cell adhesion).

Despite the wide applicability of HINs and the plethora of proposed algorithms in the literature, there is still a lack of (a) open-source, scalable implementations of these methods, and (b) tools to facilitate their use by non-experts. Also, implementing metapath-based analysis of HINs on top of a graph database, such as Neo4j, requires significant programming skills and familiarity with the system's native query language; also, certain important features of Neo4j, including distributed execution, are only available in the Enterprise Edition. As a first attempt to fill this gap, we have recently developed SPHINX [2]; however, SPHINX is mainly tailored to similarity search and does not offer parallel and distributed execution that is required to scale to larger HINs.

In this work, we introduce SciNeM[2] (Data Science tool for heterogeneous Network Mining), an open-source[3] tool that offers a wide range of functionalities for exploring and analysing HINs and utilises Apache Spark for scaling out through parallel and distributed computation. SciNeM provides an intuitive, Web-based user interface to build and execute complex constrained metapath-based queries and to explore and visualise the corresponding results. Under the hood, all the supported state-of-the-art HIN analysis types have been implemented in a scalable

[1] Note that some edges have been added for presentation purposes and may not reflect real relationships.

[2] http://scinem.imsi.athenarc.gr
[3] https://github.com/schatzopoulos/SciNeM

manner supporting the distributed execution of analysis tasks on computational clusters. SciNeM has a modular architecture making it easy to extend it with additional algorithms and functionalities. Currently, it supports the following operations, given a user-specified metapath: ranking entities using a random walk mode, retrieving the top-k most similar pairs of entities, finding the most similar entities to a query entity, and discovering entity communities.

2 SYSTEM OVERVIEW

2.1 Architecture & Functionalities

Figure 2 illustrates the key components of SciNeM's architecture, as well as the data flow between them. All (back-end) components have been implemented on top of Apache Spark to allow scalable execution on computational clusters. In the following paragraphs, we elaborate on their functionality and implementation.

2.1.1 Distributed HIN Storage. This is SciNeM's main storage layer. It is responsible for the storage of all HIN data and it is based on a Hadoop Distributed File System (HDFS) hosted on the storage media of the underlying computational cluster. Each HIN consists of a set of files including (a) a *schema file*, that describes the HIN node types and the types of relationships between them (compatible with Cytoscape's Elements JSON format[4]), (b) *node files* in TSV format containing data attributes for the nodes of each type, and (c) *relationship files* that define the edges of the network. User-created HINs can be uploaded to this storage layer via the Web front-end.

2.1.2 HIN Transformation. Most metapath-based analysis types, like those discussed in Section 2.1.3, require a common preprocessing step that transforms the initial heterogeneous network to a homogeneous (or bipartite) one. This network is essentially a *view* of the HIN containing only the nodes of the first (or the first and last, respectively) entity type in the metapath and having one edge for each metapath instance connecting these entities. Further analysis is performed on the aforementioned HIN view.

The HIN Transformation component implements this preprocessing step. It takes as input a user-defined metapath and a set of constraints and identifies all pairs of nodes that are connected based on this constrained metapath. For each pair, it also captures the number of metapath instances that connect the corresponding nodes.

Since the calculation of the metapath-based view is a computationally intensive task, special care was taken for the efficient implementation of this component. The core of transformation is calculated using matrix multiplication between the adjacency matrices defined by the relations of the given metapath. Specifically, our approach is based on the work in [6] but extends it by utilising sparse matrix representations. Since the order of multiplications significantly affects the performance of the whole processing, we adopt a dynamic programming approach that estimates the optimal ordering taking into consideration the computational cost of sparse matrix multiplications introduced in [4]. This modification offers significant speedups in many cases. In addition, the implementation of this component utilises Apache Spark, thus taking advantage of parallel and distributed computing.

2.1.3 Metapath-based analysis. This component implements a range of metapath-based mining tasks for HINs. In particular, state-of-the-art methods for entity ranking, similarity join, similarity search, and community detection are implemented, as explained below.

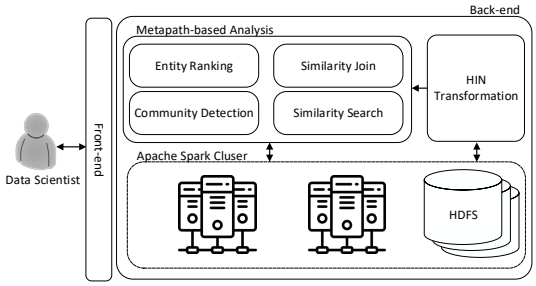


Figure 2: Architecture of SciNeM.

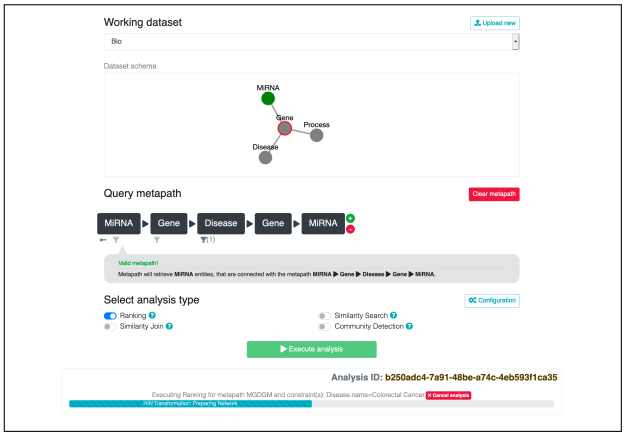
Given a particular constrained metapath, the *Entity Ranking* component estimates the significance of entities according to a random walk model applied to the corresponding HIN view [6]. In particular, the PageRank score of each node in the HIN view is calculated, and the corresponding entities are ranked based on these scores. The intuition is that this procedure brings as top-ranked results nodes which are well-connected inside the metapath-based view, i.e., nodes that correspond to entities which are important according to the semantics of the selected constrained metapath (this is why so many other nodes connect to them). To guarantee scalability, a high-performance Spark-based ranking component has been developed, allowing the analysis of very large HINs.

The *Similarity Join* component identifies the most similar pairs of nodes based on the way they are linked with other nodes when considering a particular (possibly constrained) metapath. As this type of analysis is computationally intensive, SciNeM leverages Locality Sensitive Hashing [3] (LSH) using Bucketed Random Projection to prune expensive similarity calculations. For each node, a feature vector is constructed based on its connectivity on the metapath-based HIN view. These vectors are then hashed into buckets, so that vectors that are similar end up in the same bucket with high probability. A similar approach is also followed by other relevant works (e.g., [11]).

The *Similarity Search* component detects nodes that are similar to a given query node. The notion of similarity used is the same as the one used by the Similarity Join component. In more details, SciNeM performs an approximate nearest neighbors search using the Euclidean Distance to determine (dis)similarity between nodes. Moreover, the same hashing technique as in Similarity Join is used to effectively prune the search space. Furthermore, it should be noted that, to improve scalability of the performed analyses, the Similarity Search and Join components have been implemented based on Apache Spark.

Finally, the *Community Detection* component identifies communities (i.e., clusters) of interacting nodes/entities based solely on the structural properties of the selected metapath-based HIN view, that is produced by the HIN Transformation component (see Section 2.1.2). The analysis is based on the Label Propagation Algorithm (LPA) [5], which is a popular community detection approach that requires no a priori knowledge about the network's structure. It is based on propagating labels throughout the network and forming the communities following the intuition that labels will be trapped and become dominant in clusters of densely connected nodes. Although this type of analysis is less intensive

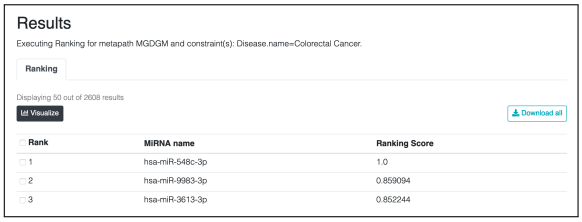
[4]https://cytoscape.org/



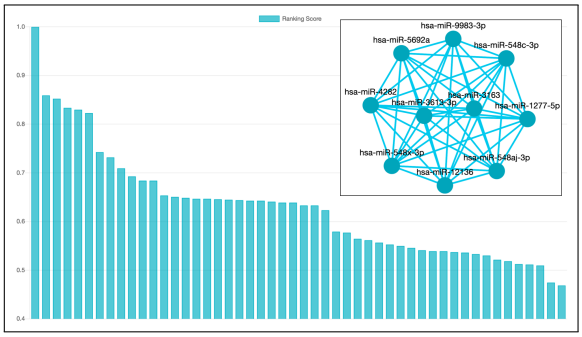
(a) Analysis task submission form.

(b) Constraint selection pop-up window.

Figure 3: Screenshots from sumbitting a Ranking analysis on the BIO dataset, using the MGDGM metapath with the constraint D.name='Colorectal Cancer'



(a) Ranking results.



(b) Visualisations of ranking results.

Figure 4: Screenshots from the results of the analysis of Figure 3.

than other approaches (e.g., Fast-Greedy, Infomap), for large networks it requires significant computational power and has a very large memory footprint. This is why the corresponding component of SciNeM takes advantage of a Spark-based, distributed implementation of the algorithm.

2.1.4 Web front-end. SciNeM's Web UI supports determining and executing metapath-based analysis tasks. These can be executed on already available HINs or new ones uploaded by the user. A *visual wizard* used to determine the details of the desired analysis tasks lies at the core of this component (see also Section 2.2). The front-end was implemented using React[5] JS library assisted with Redux[6] state container for efficient state management. Graph visualisations (e.g., HIN schema visualisation for the query builder) were implemented using the Cytoscape JS library.

2.2 User Interface

Figure 3a presents a screenshot of SciNeM's analysis task submission form. To perform a new analysis, the user first selects an existing HIN from the corresponding drop-down menu or uploads a new one. The latter requires uploading a single compressed file that contains the files described in Section 2.1.1[7].

After selecting the input HIN, the user specifies the metapath to be used for the analysis and the desired constraints. To assist the user in selecting metapaths, an interactive version of the schema of the HIN is displayed in the submission form. The user can either click on the entity types (nodes) of the schema to incrementally build the desired metapath, or add extra entity types by selecting them from a drop-down list after clicking on the green button located at the end of the currently selected sequence. To select the desired constraints, the user can click on the filter icon located below the involved entity type. A pop-up window will appear on the screen (see Figure 3b). The user selects the desired constraints and then hits the 'Save' button.

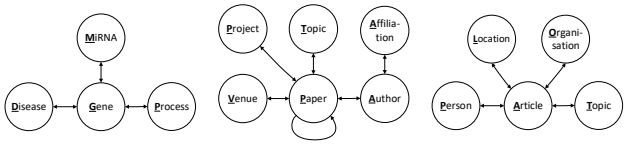
Finally, the user selects the types of analysis to be performed (multiple can be selected simultaneously) and clicks on the 'Execute analysis' button[8]. A progress bar appears in the screen (see at the bottom of Figure 3a) to monitor the status of the execution. Moreover, a unique identifier is assigned to each analysis so that the user can return to the analysis using the option 'Reattach to analysis' from SciNeM's navigation bar.

After the analysis is completed, the results appear in a tabular form (see Figure 4a). The user can browse them or select to download all or part of them. She also has the option to select some of them to create a condition file, i.e. a special file in JSON format that encodes them into a set of constraints that can be used in a later analysis. In particular, after creating a condition file the user can provide it as input in a later analysis by clicking on the 'Load from file' button of the constraints pop-up window (see Figure 3b). Essentially this creates a mechanism to use the results of an analysis as input to a subsequent one.

Finally, apart from the tabular results, SciNeM also provides a set of visualisations. The user can click on the 'Visualize' button, located above the list of results, and select the visualization type to display (for the cases for which more than one visualization type is provided). Figure 4b displays examples of such visualisations, in particular a bar chart showing the distribution of ranking scores in the top ranking results of Figure 4a and a graph showing the part of the corresponding metapath-based HIN view that contains the top-10 results (the node sizes are based on the corresponding ranking scores).

[5]https://reactjs.org/
[6]https://redux.js.org/
[7]Details can also be found in SciNeM's dataset upload page.

[8]It should be noted that for all similarity search analysis tasks the user should also determine the search entity before starting the execution.



(a) BIO (b) DBLP-Ext (c) GDELT

Figure 5: Schema definitions of pre-loaded HINs.

3 DEMONSTRATION

During the demonstration, the audience will have the opportunity to become familiar with the concepts of metapath-based analysis in HINs and to interact with SciNeM's user interface exploring its functionalities. The members of the audience will be able to execute their own analysis tasks and, if needed, to upload their own HINs. However, to facilitate examining SciNeM's capabilities, three datasets have been already prepared and made available:

- *BIO.* It contains data about the involvement of genes in biological processes and diseases (based on GeneOntology [1, 10] and DisGeNET[9], respectively). It also contains data about the suppression of genes by miRNAs (provided by MR-microT[10]). It includes 4 entity types (see Figure 5a), containing a total of 61, 177 nodes and 4, 190, 808 edges.
- *GDELT.* It contains data for news articles and associated entities collected by the GDELT project[11]. In particular, we have collected articles published in 2019 from BBC and CNN. GDELT consists of 5 entity types (see Figure 5c) totaling 245, 950 nodes and 6, 523, 924 relationships.
- *DBLP-Ext.* This HIN contains bibliographic data from the DBLP Citation Dataset of AMiner [9] enriched with european project data from the Cordis project[12]. It contains 6 entity types with 12, 152, 816 nodes and 190, 998, 307 relationships. DBLP-Ext's schema is presented in Figure 5b.

Based on these HINs, four indicative scenarios have been prepared for demonstration. Short descriptions of them follow:

Scenario 1: Important miRNAs for a disease (Ranking). Although the involvement of genes in biological processes and diseases is relatively well-studied, this is not the case for the role of miRNAs. Yet, it is possible to reveal a miRNA's role based on the list of genes it suppresses. Using SciNeM on the BIO dataset, a member of the audience can reveal miRNAs having important role in 'Colorectal Cancer' by selecting to rank miRNAs based on the *MGDGM* metapath using the *D.name = 'Colorectal Cancer'* condition. Highly ranked entities have large centrality in the corresponding HIN view, thus they are highly connected through metapath instances that satisfy the determined condition about the disease of interest. A search in PubMed reveals that there are various articles mentioning in their abstract and/or title both the top retrieved miRNA ('miR-548c') and disease of interest.

Scenario 2: Similar venues to a given one based on the topics of their published papers (Similarity Search). A member of the audience is interested in finding similar venues to the "Very Large Data Bases" (VDLB) conference, according to the topics of their recent papers. As a result, she selects to perform a similarity search on the DBLP-Ext dataset using the *VPTPV* metapath and the constraint $P.year > 2000$. The top results include very relevant venues like the "Int'l Conference on Data Engineering" (ICDE) in the first position and the "Int'l Conference on Management of Data" (SIGMOD) in the second position.

Scenario 3: Communities of organizations based on article mentions (Community Detection). A member of the audience is interested in revealing clusters of related organizations (e.g., governmental institutions, companies) based on their mentions in the news articles of an international network source like CNN. To do so, she chooses to perform community detection on the GDELT dataset using the *OAO* metapath with the *A.source* ="cnn" constraint. The results contain various interesting communities; e.g., the one with $id = 111$ consists of 3 institutions having an agenda related to climate change ("UN Intergovernmental Panel on Climate", "European Union Copernicus Climate Change Programme", "World Meteorological Organization"), whereas the one with $id = 394$ includes 7 institutions involved in politics in India.

4 CONCLUSION

We demonstrated SciNeM, an open source, high-performance and scalable online data science tool that facilitates metapath-based analysis of HINs. Its intuitive user interface aids non-experts to perform a variety of HIN analysis tasks such as metapath-based ranking, similarity join, similarity search, and community detection. Finally, SciNeM's users may upload their own HIN datasets to analyse, however the tool also provides pre-loaded datasets for demonstration reasons.

ACKNOWLEDGMENTS

This work was partially funded by the EU H2020 project SmartDataLake (825041).

REFERENCES

[1] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig, M. A. Harris, D. P. Hill, L. Issel-Tarver, A. Kasarskis, S. Lewis, J. C. Matese, J. E. Richardson, M. Ringwald, G. M. Rubin, and G. Sherlock. 2000. Gene Ontology: tool for the unification of biology. *Nature Genetics* 25, 1 (2000), 25–29.

[2] S. Chatzopoulos, K. Patroumpas, A. Zeakis, T. Vergoulis, and D. Skoutas. 2020. SPHINX: A System for Metapath-based Entity Exploration in Heterogeneous Information Networks. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB 2020).*

[3] P. Indyk and R. Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proc. of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98).* 604–613.

[4] D. Kernert, F. Köhler, and W. Lehner. 2015. SpMacho - Optimizing Sparse Linear Algebra Expressions with Probabilistic Density Estimation. In *Proc. of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium.* 289–300.

[5] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* 76, 3 (2007), 036106.

[6] C. Shi, Y. Li, P. S. Yu, and B. Wu. 2016. Constrained-meta-path-based ranking in heterogeneous information network. *Knowl. Inf. Syst.* 49, 2 (2016), 719–747.

[7] C. Shi, Y. Li, J. Zhang, Y. Sun, and P. S. Yu. 2017. A Survey of Heterogeneous Information Network Analysis. *IEEE TKDE* 29, 1 (2017), 17–37.

[8] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu. 2011. PathSim: Meta Path-Based Top-K Similarity Search in Heterogeneous Information Networks. *PVLDB* 4, 11 (2011), 992–1003.

[9] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su. 2008. ArnetMiner: Extraction and Mining of Academic Social Networks. In *Proceedings of the 14th ACM SIGKDD.* ACM, 990–998.

[10] The Gene Ontology Consortium. 2018. The Gene Ontology Resource: 20 years and still GOing strong. *Nucleic Acids Research* 47, D1 (11 2018), D330–D338.

[11] Y. Xiong, Y. Zhu, and P. S. Yu. 2015. Top-k Similarity Join in Heterogeneous Information Networks. *IEEE Trans. Knowl. Data Eng.* 27, 6 (2015), 1710–1723.

[9] https://www.disgenet.org
[10] http://diana.imis.athena-innovation.gr/DianaTools/index.php?r=mrmicrot/
[11] https://www.gdeltproject.org
[12] https://cordis.europa.eu

IMCF: The IoT Meta-Control Firewall for Smart Buildings

Soteris Constantinou
University of Cyprus
2109 Nicosia, Cyprus
constantinou.sotiris@cs.ucy.ac.cy

Antonis Vasileiou
University of Cyprus
2109 Nicosia, Cyprus
avasil06@cs.ucy.ac.cy

Andreas Konstantinidis
Frederick University
1036 Nicosia, Cyprus
com.ca@frederick.ac.cy

Panos K. Chrysanthis
University of Pittsburgh
Pittsburgh, PA 15260, USA
panos@cs.pitt.edu

Demetrios Zeinalipour-Yazti
University of Cyprus
2109 Nicosia, Cyprus
dzeina@cs.ucy.ac.cy

ABSTRACT

In this demonstration paper, we present an innovative *IoT Meta-Control Firewall* (*IMCF*), which allows users to schedule their IoT devices in smart buildings (e.g., heating, cooling, lights) in order to reach some long-term energy consumption objective (e.g., consume less than 400 kWh in December) while, at the same time, retaining high levels of user convenience (comfort). IMCF internally deploys an AI-inspired Energy-Planner (EP) algorithm that exploits domain-specific operators to balance the trade-off between convenience and energy consumption. Our framework then filters the rules of users in a way that these do not conflict with the long-term objectives (i.e., like a network firewall). We demonstrate IMCF using a prototype system we have developed in the Laravel PHP web framework using the open Home Automation Bus (OpenHAB), the Linux crontab daemon and Anyplace for building modeling. In our demonstration scenario, attendees will be able to observe the execution and benefits of IMCF on a graphical dashboard using pre-configured or custom-made Meta-Rule-Table profiles.

1 INTRODUCTION

Internet of Things (IoT) refers to a large number of physical devices being connected to the Internet that are able to "see", "hear", "think", "react", perform tasks, as well as communicate with each other using open protocols [8]. According to Gartner[1], it is expected that the number of IoT devices per house will increase to more than 500 smart devices by 2022. Many IoT devices also enable the execution of *Rule Automation Workflows (RAW)*, which span from simple predicate statements to procedural workflows capturing a smart actuation pipeline in tools like IFTTT [5], controlling Philips Hue lights, BMW i3 EVs or Daikin A/Cs [7][4], Apilio.io, or Apple Automation.

RAW aim to meet the convenience level of users under specific conditions (e.g., *"warm house to 22°C if cold or preheat Electric Vehicle when approaching"*). In the simplest case, a user expresses preferences manually through a vendor-specific smartphone app / integrated app (e.g., see Fig. 1). This process requires continuous attention by custodians, making it a cumbersome process that generates erroneous executions and that clearly calls for more automated (i.e., *"smarter"*) approaches.

[1]Gartner Inc., URL: https://tinyurl.com/ycrxsmy6

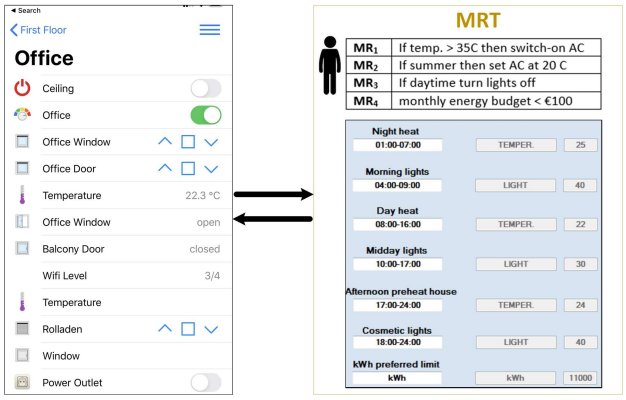


Figure 1: The openHAB bridge gives the privilege to users to adapt a Meta-Rule-Table (MRT) profile as necessary

One of the most straightforward approaches to achieve a smarter RAW is to utilize the so-called *trigger-action* model. Users control the behavior of an IoT by specifying triggers (e.g., *"if it is sunny outside"*) and their resultant actions (e.g., *"turn off the lights"*). Because of its conceptual simplicity, the trigger-action model (a.k.a. Event-Condition-Action) has attracted significant attention with ifttt.com (*"If This Then That"*) becoming one of the first large-scale deployments. Services like Apilio expanded the expressiveness of the RAW with Boolean predicates (e.g., conjunctions) and Apple Automation [3] even introduced procedural programming constructs, like variables, while loops, if statements and functions to advance RAW actuations.

However, none of the above RAW technologies enables individuals or group of users to express their convenience (comfort) preferences while achieving some long-term objective. Similarly, prior research [6] was mainly concerned with improving comfort levels of HVAC system but not long-term energy planning targets. In our scenario, the long-term objective relates to *energy consumption* (e.g., in kWh), which is motivated by European's Commission calls for a climate-neutral Europe by 2050[2]. Particularly, we aim to consume energy more intelligently and within margins we define as persons or group of users.

In this demo we present an innovative system, coined the *IoT Meta-Control Firewall (IMCF)* [2], which aims to fill the gap of manual RAW tuning to reach the energy consumption targets. The user (or group of users) start out by defining a vector of RAW rules, dubbed *MRT*, and an *Energy Consumption Profile*, dubbed *ECP*. The high-level objective is to identify among all MRT rules

[2]EU 2050 long-term strategy, `https://tiny.cc/9wu8iz`

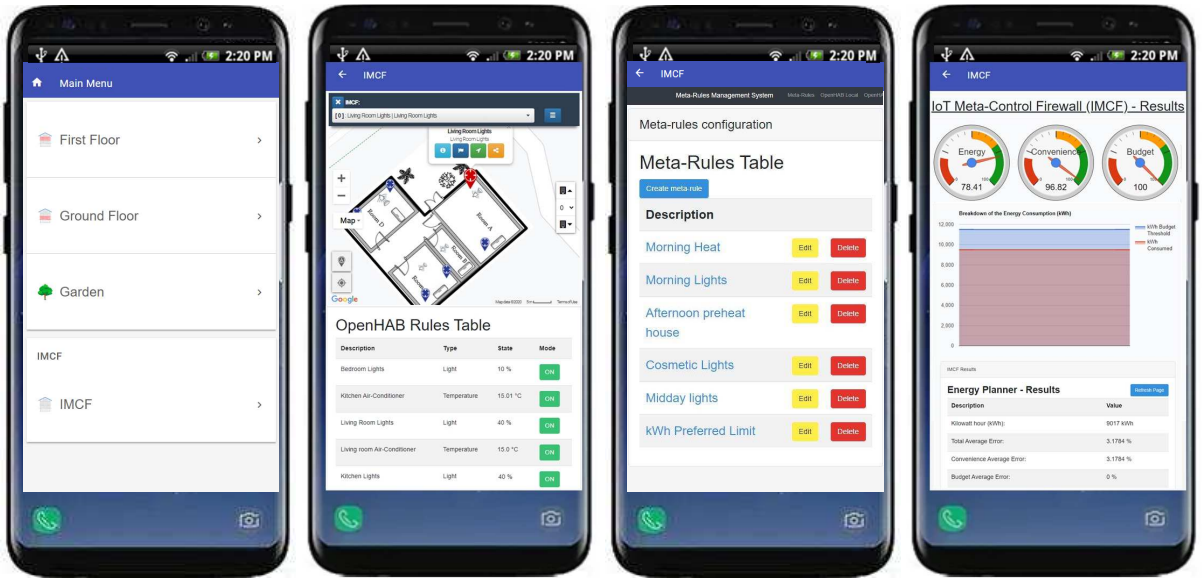


Figure 2: IMCF Graphical User Interface: Integration of the IMCF Software Library in the openHAB Home Automation Stack. From left to right: a) Interactive and Automated Menu; b) Dashboard for smart space current state linked with Anyplace Viewer; c) Meta-Rule-Table Configurator; and d) IMCF Results.

the ones that must be dropped so that the user stays within the desired energy budget according to the *ECP* history (e.g., consume less than 400 kWh in December). For this purpose, it utilizes an intelligent search algorithm, coined *EP (Energy Planner)*, which goes over the exponentially large search space of $\sum_{r \leq n} r$-combinations (where $n = |MRT|$), yielding quickly the rules to be dropped. Particularly, IMCF is composed of an intelligent energy amortization process and an AI algorithm for balancing the trade-off between convenience and energy consumption, and satisfying the RAW pipelines of users in a way that these do not conflict with the long-term objectives.

Considering several system implications, IMCF can be implemented in various ways such as the following: (i) a *cloud meta-service*, which guides the IFTTT *cloud service* that in turn controls a *local controller* (e.g., Linux, as in the case of OpenHab), which controls the sensors/actuators (e.g., A/C unit); or (ii) a *local controller* (filtering rules out at the network level on the local controller).

2 THE IOT META-CONTROL FIREWALL (IMCF) OVERVIEW

In this section, we describe a prototype system we have developed for IMCF using the open Home Automation Bus (OpenHAB)[3], the Linux crontab daemon, Anyplace for building modeling [1], as well as the Laravel PHP web framework following the model–view–controller architectural pattern.

We start out with a discussion of the system architecture, followed by the IMCF algorithm, and then describe the Graphical User Interface (GUI) we have developed. The GUI integrates directly into OpenHAB's mobile and web Panel view for both interactive management of IoT and automated management of Energy-aware MRT pipelines using the EP described below.

[3]OpenHAB, `https://openhab.org`

2.1 System Architecture

Our system architecture comprises of the following components: (i) a full-fledge local controller implemented inside the openHAB stack, which is a smart home management software; and (ii) IMCF, which is the software system that encapsulates the complete application logic of the energy management stack we propose along with the respective user interfaces.

Local Controller (LC): is a java-based system that can be installed on a micro device, like a Linux Raspberry PI, running on the local network of a user. The LC will be in direct communication with the IoT devices (i.e., *Things (TG)*) to instruct them based on the preferences registered by a user. A user will typically download the openHAB smartphone *application (APP)*, for iOS or Android, and interact with TG through LC. For the implementation of LC we decided to extend the openHAB stack, which is a vendor and technology-agnostic open source automation software for smart home that provides a rich ecosystem of bridges through which a user can interact directly with IoT devices (e.g., Daikin Smart A/C, Phillips HUE lights) both locally and remotely. This gives us the benefit to achieve maximum IoT market compatibility as the integration of IoT is always an immense challenge.

To realize the operation of LC consider, for example, a user inside his smart space that uses an APP to increase the temperature of an A/C from 21 to 25 degrees Celsius (see Figure 2a-b). This manual interaction goes directly to LC that eventually communicates with TG (on older units this is typically refers to unencrypted http communication channels, either http querystring or in some cases JSON web 2.0 interactions). When a user's APP is outside a smart space, the network firewall and Network Address Translation (NAT) will obviously not let this user interact with LC. As such, the user's APP connects to the *Cloud Controller (CC)*, which is a server on the public Internet that communicates and controls LC remotely. The complete picture can tentatively be complemented by a *Cloud Meta-Controller (CMC)*, like IFTTT, which can enable the user to configure and run various custom rules. CMC would in this case interact with CC that

Table 1: Evaluating our system prototype with respect to Energy Consumption (F_E) and Convenience Error (F_{CE})

Time Duration	Energy Consumption (F_E)	Convenience Error (F_{CE})
Week	130.64 kWh	2.35%

Table 2: Individual Resident Convenience Error (F_{CE})

Users	Convenience Error (F_{CE})
Father	0.8006%
Mother	0.7899%
Daughter	0.7595%

would in turn interact with LC that would eventually interact with TG, all under the *manual* control of the user APP.

The IMCF Component: is a software extension to LC we have implemented to enable the adaptation of convenience preferences to meet the long-term energy planning targets of individuals or group of individuals. It has been developed in a way that encapsulates the implementation of the EP algorithm but also the GUI and storage necessary to allow the user to interact with the system. EP is implemented as a JAVA library, which takes the user configurations from a local MariaDB persistency layer. The storage layer is populated by the user using the APP, which has been configured in a way to integrate seamlessly the *MRT* rule definition process through a web-based GUI (see Figure 2c,d). The GUI code is written in the Laravel PHP web framework, as well as JavaScript and HTML. The GUI code execution relies on a webserver supporting PHP while for the IMCF EP library a cron job daemon is assumed (available on Linux) that reliably invokes the Energy Planning in fixed time intervals (e.g., every few minutes). In case devices have to be turned on or off, the IMCF system has the following options in our system:

- *Binding-mode*, where IMCF exploits the rich ecosystem of bridges available on the openHAB open source project to interact with local devices. We use this as the default mode, as it allows our platform to scale up to a wide spectrum of IoT devices.
- *Extended mode*, where IMCF implements locally the custom instructions for enabling and disabling the various TG devices in the smart space of a user.
  Given that many of the IoT communications are unencrypted, this can be easily captured by deep packet analyzers like Wireshark. Moreover, to avoid any additional CMC, CC or LC interactions with TG, we also configure the LC network firewall with the `iptables` command to disable TCP flows to designated TG devices on the local network. In this case, IMCF works actually as a real network firewall by blocking all outgoing traffic from LC to TG.

Case scenario: We have deployed an instance of our real prototype system for a family of three persons for one week. Particularly, we allowed each person to configure their personal preferences using the Mobile APP that interacts with an IMCF-LC node on a Linux VM on our datacenter described earlier. Particularly, each individual resident entered approximately three different meta-rules according to their personal preferences. One of them have set the weekly energy consumption (kWh) limit to 165kWh. This results in configuration data of approximately 65 bytes / user stored in the MariaDB persistency layer. In order to measure the environmental parameters (i.e., temperature, light) we use data from the open weather forecast API. We measure the performance of the proposed EP framework in regards to Energy Consumption (F_E) and Convenience Error (F_{CE}). The F_E and F_{CE} results for our evaluation are summarized in Table 1. In respect to F_{CE} our observation is that EP is indeed an efficient approach for retrieving great user satisfaction, as it performs in 4 seconds on average with $F_{CE} \approx 2.35\%$. Table 2 demonstrates for each individual resident their own Average Convenience Error values in respect with their configured meta-rules, showing both a consistent and high satisfaction close to 99.2% for all residents. Another observation is that $F_E \approx 130.64$ kWh is within the preferred budget limit as pre-configured by the user, and the system behaves correspondingly to what we observed in the simulations. Please note, this particular framework can be equally utilized in various cases such as CO_2 emission deduction, or any other scenario type that requires a planning to conserve some kind of resources.

2.2 The *IMCF* algorithm

The *IMCF* algorithm is composed of two subroutines: (i) the *Amortization Plan (AP)*; and the (ii) the *Energy Plan (EP)*. The amortization plan is responsible for calculating the maximum energy budget constraint (coined E_p) through a pre-selected amortization formula. Then an artificial intelligence approach is executed every t seconds (e.g., hourly, daily, monthly, yearly preference) over a time period p (i.e., the complete duration of the execution) for generating an energy plan solution s^* for optimizing the Convenience Error

$$\min F_{CE} = \sum_{k=1}^{t} \left(\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{D} ce_j(MR_i)\right), \tag{1}$$

where ce_j is the difference between the desired output value $\Omega_i^j \in \mathfrak{R}$ of a rule set by a user (temperature or light intensity level) and the actual value $O_i^j \in \mathfrak{R}$ set by the controller, given by: $ce = |\Omega_i^j| - |O_i^j|$.
Subject to satisfying the Energy Consumption $F_E(s^*) \leq E_p$, where:

$$F_E = \sum_{k=1}^{t} \left(\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{D} e_j(MR_i)\right), \tag{2}$$

E_p is total available energy budget for the complete period p during which the execution of our algorithm takes place, N is the total number of meta-rules, D the set of all IoT devices and e_j is the energy consumption of device j given the action defined by output O_i^j of meta-rule MR_i, given by:

$$E = \begin{cases} e_j, & \text{if } O_i^j \text{ is executed} \\ 0, & \text{otherwise} \end{cases},$$

where e_j is the energy cost of device j for MR_i.

Amortization Plan (AP) Algorithm. The $AP()$ subroutine is initially executed for calculating the energy budget constraint E_p, subject to a monthly residence *Energy Consumption Profile ECP*. There are several amortization strategies that can be used, such as Linear Amortization Formula, Balloon Linear Amortization Formula, and ECP-based Amortization Formula. Given that our approach requires no training data and only a primitive MRT preference profile, this can be easily integrated in smart actuations platforms.

Energy Plan (EP) Algorithm. An energy plan solution is a vector $s =< s_1, \ldots, s_N >$. A vector component s_i represents a

meta-rule in the meta-rule-table MRT, where $s_i = 0$ means ignoring meta-rule at position i of table MRT and $s_i = 1$ means adopting meta-rule at position i. We have adopted a hill-climbing algorithm, an iterative local search heuristic, which doesn't require a learning history (like respective Machine Learning techniques), doesn't require a target function (e.g., like A*) and is straightforward to be implemented in a resource-constraint setting like local smart controllers (e.g., Raspberry). At the beginning of the local search heuristic an initial solution s^* is developed that will specify the initial state of the algorithm either randomly or deterministically. For the optimization step, a hill-climbing local search heuristic is utilized for local optimization with neighborhoods that involve changing up to k components of the solution, which is often referred to as k-opt. Each solution s is evaluated using the performance metrics F_E and F_{CE}. A solution s is considered better and replaces the current best solution s^* if $(F_E(s) \leq E_p)$ && $(F_{CE}(s) < F_{CE}(s^*))$. The energy planner stops when τ_{max} iterations are completed. Alternatively, the algorithm can iterate until $\nexists s | F_{CE}(s) < F_{CE}(s^*)$.

2.3 Graphical User Interface (GUI)

Our prototype GUI provides all the functionalities for a user participating in IMCF. The GUI is divided into a Meta-Rule-Table interface and the OpenHAB Rules Table, respectively as shown in Figure 2b-c. The Meta-Rules interface prompts users to define kWh preferred limit, temperature and light values for any configured time slots. The OpenHAB Rules Table records are retrieved through the OpenHAB Rest API system consisted of smart device sensor measurements installed and pre-configured in a building. These rule combinations are used by the AI Energy Planner algorithm to satisfy the user needs, while keeping the balance between convenience and energy consumption.

At a high level, our GUI enables the following functions: (i) record OpenHAB item measurements/values on local storage and present those on a table; (ii) configure various meta-rules in regards of kWh limit, temperature and light values; (iii) operate IMCF framework and get an efficient execution considering user satisfaction along with balanced F_{CE} and F_E.

3 DEMONSTRATION SCENARIO

During the demonstration, the attendees will be able to appreciate the key components in IMCF, as well as the adaptability and the performance of our propositions (see Figure 2).

3.1 Demo Artifact

We have implemented a prototype of IMCF as a standalone program that is loaded on a Linux-based local controller. Particularly, we implemented a graphical user interface in the Laravel PHP web framework, following the MVC architectural pattern where a user has the privilege to upload a MRT profile that is stored on the filesystem of the Linux device. A cron job has been programmed in order to initiate our energy planner every few seconds. Every time our program runs, it decides whether certain local IPs have to be banned to satisfy user's profile by interacting with the Linux firewall using the iptables commands.

3.2 Demo Plan

The conference attendees will have the opportunity to interactively engage with the OpenHAB and MRT user profile website by setting up configurations through a smartphone. We will preload a variety of synthetic and web-accessible rules to the MRT user profile website back-end. The loaded rules will capture the structure and needs of real residential data and will be very useful to visually demonstrate how the IMCF algorithm works in real time through the OpenHAB application.

The main objective of our intelligent algorithm is to identify among an exponentially large search space of MRT rule combinations the ones that must be dropped so that the user stays within the desired energy budget. In order to present the benefits of our propositions to the attendees, we will provide visual cues that will enable the audience to understand the performance benefits (i.e., CPU time) and the negligible reduction in energy consumption and convenience error we have observed during the experiments.

We will also provide conference attendees the opportunity to create custom MRT profiles through the website. Our hypothesis is that many data engineering researchers and practitioners would feel more comfortable to formulate MRT profile predicates, as opposed to be limited within the boundaries of well-defined MRT templates provided. We will provide participants the possibility to upload an actual building plot through Anyplace. The *OpenHAB* interface will allow the attendees to rapidly visualize the result-sets on a smartphone, using fancy charts (pie, bar, etc.) when the rules get enabled or disabled by the proposed firewall, respectively. Our particular aim here will be to describe how the IMCF structure, residing on the OpenHAB, will be accessible to enable/disable rule services.

4 FUTURE WORK

In the future, we plan to further investigate multiple energy planners with conflicting interests but also to investigate the so-called IMCF-Cloud extensions that will enable IMCF to operate as a CMC controller in the cloud. We also aim to look at CO2 reductions methods with algorithms geared towards the environment. Finally, we aim to investigate power workload identification methods for power-hungry devices (e.g., white devices, electric vehicles, heating) and how to reschedule those workloads in a environmental friendly manner.

REFERENCES

[1] Marileni Angelidou, Constantinos Costa, Artyom Nikitin, and Demetrios Zeinalipour-Yazti. 2018. FMS: Managing Crowdsourced Indoor Signals with the Fingerprint Management Studio. In *19th IEEE International Conference on Mobile Data Management*. pp. 288–289. https://doi.org/10.1109/MDM.2018.00054

[2] Soteris Constantinou, Andreas Konstantinidis, Demetrios Zeinalipour-Yazti, and Panos K. Chrysanthis. 2021. The IoT Meta-Control Firewall. In *37th IEEE International Conference on Data Engineering*. IEEE Computer Society, April 19 - April 22, Chania, Crete, Greece, 12 pages (accepted).

[3] X. Meng, W. Cong, H. Liang, and J. Li. 2018. Design and implementation of Apple Orchard Monitoring System based on wireless sensor network. In *IEEE International Conference on Mechatronics and Automation*. 200–204. https://doi.org/10.1109/ICMA.2018.8484350

[4] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. 2017. An Empirical Characterization of IFTTT: Ecosystem, Usage, and Performance. In *Internet Measurement Conference*. pp. 398–404.

[5] Steven Ovadia. 2014. Automate the Internet With "If This Then That" (IFTTT). *Behavioral & Social Sciences Librarian* 33, 4 (2014), 208–211. https://doi.org/10.1080/01639269.2014.964593 arXiv:https://doi.org/10.1080/01639269.2014.964593

[6] Daniel Petrov, Rakan Alseghayer, Panos K. Chrysanthis, and Daniel Mosse. 2019. Smart Room-by-Room HVAC Scheduling for Residential Savings and Comfort. In *The 10th Intl. Green and Sustainable Computing Conf.* pp. 1–7.

[7] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. *In ACM CHI Conference on Human Factors in Computing Systems* (2016), 3227–3231.

[8] Lina Yao, Quan Z. Sheng, and Schahram Dustdar. 2015. Web-Based Management of the Internet of Things. *IEEE Internet Computing* 19, 4 (2015), pp. 60–67.

BBoxDB Streams: Distributed Processing of Real-World Streams of Position Data

Jan Kristof Nidzwetzki
FernUniversität Hagen
Hagen, Germany
jan.nidzwetzki@studium.fernuni-hagen.de

Ralf Hartmut Güting
FernUniversität Hagen
Hagen, Germany
rhg@fernuni-hagen.de

ABSTRACT

BBoxDB Streams is an extension of the key-bounding-box-value store BBoxDB. The extension allows the handling of multi-dimensional data streams. Multi-dimensional streams consist of *n*-dimensional elements, such as position data (e.g., two-dimensional positions of cars or three-dimensional positions of aircraft). In this demonstration, we show how BBoxDB Streams can be used to process data streams of position data in a distributed manner. The software allows the user to capture data streams and process continuous queries. Continuous range queries or continuous spatial joins are supported. The GUI of BBoxDB Streams shows the query results interactively as an overlay over a map. For the demonstration, public real-world data streams with positions of aircraft and transport data are processed. Continuous range queries such as *which aircraft is currently in the area of Berlin?* or continuous spatial join queries such as *which bus drives currently through a forest?* are executed, and the results can be observed in real-time. The spatial joins are executed between the stream data and previously stored static geographical information (e.g., the polygons of roads or forests), which are fetched from the OpenStreetMap Project.

1 INTRODUCTION

Data streams consisting of position data are ubiquitous. For example, aircraft periodically broadcast their positions via *ADS-B messages* (*Automatic Dependent Surveillance–Broadcast*), and the positions of buses, trains, or ferries of public transport companies are available in real-time via the internet in GTFS format (*General Transit Feed Specification*). Processing streams containing position data is an essential topic in location-aware applications. The ability to capture data streams and the near real-time execution of queries is required to deal with the information of the stream. Data streams can contain a lot of elements, and queries can be expensive to evaluate. Therefore, a scalable solution is required to process data streams.

BBoxDB Streams is an extension of *BBoxDB* [15], which allows the efficient handling of *n*-dimensional data streams. BBoxDB streams implements a novel way to execute efficient continuous joins between dynamic elements from a data stream and static already stored *n*-dimensional big data. This capability is shown for the first time in this demonstration. Besides, the GUI of BBoxDB was enhanced to execute queries on data streams and show the results interactively. This new enhancement is shown in this demonstration for the first time. BBoxDB streams is included in BBoxDB since version 0.9.5 and licensed under the Apache 2.0 license. The software can be freely downloaded from the website of the project [3].

In this demonstration, we show two types of queries with BBoxDB Streams: (1) *continuous range queries*, and (2) *continuous spatial joins*. Continuous range queries can answer questions such as *which aircraft are inside of a specific region of the airspace?*. Continuous spatial joins can be used to join the dynamic position data of the stream with static data (e.g., geographical information). With a continuous spatial join, queries such as *which buses are closer than 10 miles to a forest?* or *which bus drives on a particular street?* can be answered.

The rest of the paper is organized as follows: Section 2 describes the key-features of BBoxDB and BBoxDB Streams. Section 3 describes our demonstration. Section 4 describes the related work. Section 5 concludes the paper.

2 BBOXDB AND BBOXDB STREAMS

The amount of data is increasingly growing. NoSQL databases like *distributed key-value stores* (DKVS) are often used to handle large amounts of data. In a DKVS, the data are assigned to the nodes of a cluster. Each node stores only a part of the whole dataset. A value is stored under a given key. Using this key, the value can be retrieved. The key is the access path to the data. A key for a one-dimensional value can be easily chosen. For data with a higher dimensionality, a key is hard to choose. Which key should be used for the geographic information about a road? Using the name of the road as the key does not help to access the data when a spatial range query is performed; the name does not contain the information where the road is located. To answer such range queries, a full data scan has to be performed; all stored tuples have to be loaded and it needs to be tested whether or not the stored value intersects with the given query rectangle. This is an expensive operation that is performed on all nodes of the distributed system.

BBoxDB was designed to solve this problem. BBoxDB is a distributed *key-bounding-box-value store* (KBVS) which supports the efficient storage and retrieval of *n*-dimensional data.

2.1 Basic Concepts of BBoxDB

BBoxDB is a distributed generic datastore optimized for the handling of *n*-dimensional big data. Values are stored as byte arrays together with a key and an *n*-dimensional bounding box as tuples. The bounding box describes the location of the *tuple* in the *n*-dimensional space. Point and non-point data are supported by BBoxDB. Tuples are grouped together in *tables* and multiple tables of the same dimensionality can be stored together in a *distribution group*. The space is split automatically to ensure almost equal-sized partitions; the data of these partitions (called *distribution regions*) are assigned to the nodes of a cluster. The tables of the same distribution group are distributed in the same way; this means the tables are stored co-partitioned, which enables the execution of efficient spatial joins. No data need to be transferred through the network; all join partners are stored on

the same node. The complete system is highly available, data can be replicated and failing nodes are handled automatically.

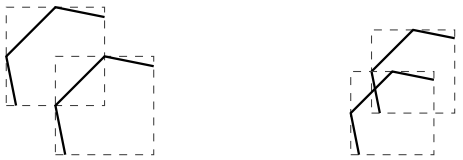
Operations: Data are stored in BBoxDB with the `put(table, key, hrect, value)` operation. As `hrect` parameter, an *n*-dimensional bounding box (a hyperrectangle) has to be specified. One-dimensional point data (as used in DKVS) can also be stored in BBoxDB. In this case, the bounding box degenerates to a point in the one-dimensional space.

Data are retrieved by the `queryByRect(table, hrect)` operation, which retrieves all tuples whose bounding box intersects with the query bounding box. The operation `join(table1, table2, hrect)` executes a spatial join between the two tables in the specified region in space.

Indexing: BBoxDB uses a two-level indexing structure that enables the efficient execution of range queries. The *global index* is used to map the distribution regions to the nodes of the cluster ($distribution\ region \rightarrow \mathcal{P}(nodes)$)[1]. The space is partitioned (split and merged) by a *space partitioner* automatically, based on the stored data [13]. Splitting and merging the space is done transparently in the background without interrupting the access to the data. BBoxDB provides multiple algorithms for the global index. The used algorithm can be specified (KD-Trees [4] or Quad-Trees [5]) when the distribution group is created. The *local index* is stored on the nodes and maps from the space to the stored tuples ($space \rightarrow tuples$). This index is implemented by an R-Tree [7] which is stored on the nodes.

User-defined filters: BBoxDB is a generic data store; the stored values are a plain array of bytes. BBoxDB does not understand the semantics of the stored data. Therefore, operations are executed primarily on the bounding boxes of the tuples. *User-defined filters* [14] (UDFs) can be used to decode the bytes of a value (e.g., GeoJSON encoded data) and to refine the bounding box based operations of the query processor. UDFs are developed by the user of the system. Only the user who has stored the data knows how to interpret the values of the data. Besides, BBoxDB ships with a collection of UDFs for common data formats. UDFs are written in Java and can use existing libraries.

One of the UDFs that have been included is capable of decoding GeoJSON data. Using this UDF, a bounding box based spatial join is refined to a spatial join on the real geometries of the values. Intersecting bounding boxes is a necessary but not sufficient criterion for a spatial join (see Figure 1).



(a) Two non-intersecting spatial objects. (b) Two intersecting spatial objects.

Figure 1: Two spatial objects (solid line) with intersecting bounding boxes (dashed line). In (a), the spatial objects do not intersect, while in (b), the spatial objects do intersect.

This UDF is used in the demonstration to refine the continuous spatial joins. The property map of GeoJSON encoded objects (see Listing 1) can also be taken into consideration in the UDF. For example, this can be used to filter streets of a specific name like *Elizabeth Street*. Queries such as *find all buses which are on a street named Elizabeth Street* become possible.

[1]When replication is used, one distribution region is mapped to multiple nodes.

2.2 BBoxDB Streams

BBoxDB is an extension of BBoxDB that allows the handling of data streams. With the extension, data streams can be captured and continuous queries can be executed. Figure 2 shows the architecture of the extension. The upper part of the figure shows the stream capturing part, the lower part the query processing part.

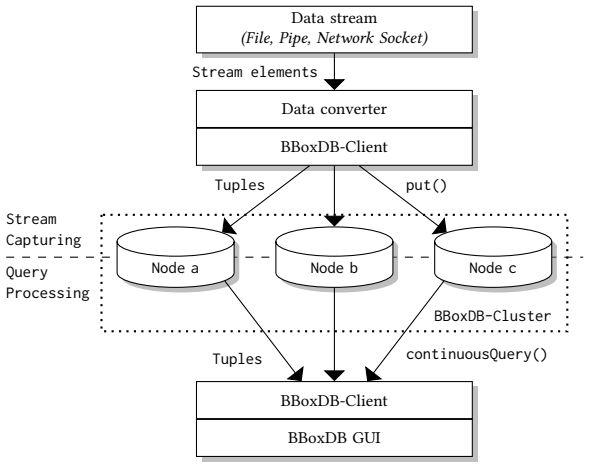


Figure 2: Handling a data stream with BBoxDB. The data stream is captured, converted into tuples, and written to the BBoxDB cluster. Afterward, the continuous queries are executed and the result can be consumed.

Stream Capturing: BBoxDB Streams captures data continuously from an input source like a *file*, a *pipe*, or a *network socket*. After a stream element is read, the element is converted into a BBoxDB tuple and sent to the BBoxDB nodes of the cluster[2]. To communicate with the BBoxDB nodes, the regular BBoxDB-Client library is used. The library manages the connection to the nodes, observes the global index, and executes the operations on the necessary nodes. Changes of the global index or the available nodes are handled. The converted stream elements are written to the BBoxDB cluster by executing the `put()` operation. The bounding box of these tuples is compared with the global index; the tuple is written to all nodes that are responsible for the region is space. On these nodes, the potential join partners are stored for a spatial join; the stream elements become co-partitioned to the already stored data, and efficient spatial joins become possible. After a node receives a tuple, the registered continuous queries are executed. The highly-available architecture is also used for the processing of the streams. Distribution groups can be stored replicated, in this case the continuous queries are also registered on multiple nodes. As the stream is written to a table, BBoxDB splits the space, updates the global index, and re-distributes uneven distributed tables automatically as described in [15, p. 20].

Query Processing: BBoxDB Streams enhances BBoxDB by two operations for the handling of continuous queries: (1) `continuousQuery(queryPlan)` and (2) `cancelQuery(id)`. The first operation registers a new query while the second operation cancels a previously registered query. The existing BBoxDB-Client

[2]BBoxDB ships with some data converter for common data formats like GTFS or ADS-B. The converter decodes the input data, calculates the bounding box and creates a tuple. Further data converters can be added easily by a user. For certain stream types, multiple elements from the stream are combined into one BBoxDB tuple. For example, the ADS-B format defines multiple message types. Three different message types have to be read to get the current data of an aircraft.

was enhanced by BBoxDB Streams to register continuous queries. The BBoxDB-Client automatically registers the queries on the required nodes of the cluster; this part of the client library was re-used. In this demonstration, the GUI of BBoxDB uses the BBoxDB-Client to register the queries and to obtain the results. These tuples are consumed by the GUI and shown as an overlay of a map. The stream handling functionality is now integrated into the regular client library. Any application can create queries on data streams and consume the results.

The query plan defines the operations of the continuous query. In the query plan, (1) the *type* of the query (range query or spatial join), (2) *transformations*, and (3) *filters* are defined. Technically, the query plan is a JSON document that is sent to the BBoxDB nodes. A helper class is available, which allows the easy and syntactically correct creation of these query plans.

Transformations allow the modification of the stream elements and the potential join partners. For example, the bounding box of an aircraft can be extended and joined with the obstacles in the airspace. Due to the enhancement of the bounding box, a possible collision is detected and reported before the aircraft and the obstacle actually collide. Also the potential collision of two aircraft can be detected. This is done by storing the data stream in a table and performing a continuous spatial join between the new stream elements and the materialized stream elements from the table.

Filters allow removing elements from the stream or from the list of potential join partners. For example, process only the aircraft with a call sign starting with `LH`. In addition, UDFs can also be used as a filter.

The complete architecture of BBoxDB and BBoxDB streams is horizontally scalable. When more static data have to be stored, the existing distribution regions can be split, and these new regions can be assigned to further BBoxDB nodes. This can also be done to process a larger stream or more continuous queries. The use of more distribution regions splits up the stream into more parts. Each part is handled by an individual node that has its own resources to capture the stream and execute the registered queries.

3 DEMONSTRATION

A cluster of five nodes is used in our demonstration, which is located at our university. Each node contains an Intel Xeon E5-2630 CPU, 32 GB of memory and four 1 TB hard disks. All the nodes are connected via a 1 Gbit/s network and running Java 8 on a 64 bit Ubuntu Linux. A notebook is used to run the GUI and to perform the demonstration.

3.1 Data Streams for the Demonstration

In this demonstration, two real-world data streams are used: (1) *The ADS-B aircraft data stream* and (2) *the Sydney transport data stream.*

An aircraft continuously broadcasts its position periodically via radio as ADS-B transmissions. The transmissions contain the position of the aircraft, the height, the call sign, and some more information. However, an ADS-B receiver captures only the transmissions in a radius of some miles around the antenna. Websites such as adsbhub.org [19] provide a service to aggregate the feeds of several individual stations into a global feed, containing the flight data of the whole world.

The government of the state of New South Wales in Australia operates the *NSW open data portal* [17]. On this portal, real-time data about buses, ferries, metros, and trains of the region are published. A GTFS encoded real-time feed of the data can be subscribed.

For our demonstration, the elements of both data streams are used and converted into GeoJSON objects. GeoJSON is a format that can be read and understand by a human (in contrast to binary-encoded GTFS data), which makes it suitable for demonstration purposes. Listing 1 contains one element from the public transport data steam after it is converted into GeoJSON. In addition to the position of the bus, further *properties* are contained which contain additional information such as the route or the speed of the vehicle.

Listing 1: Bus trip data converted into GeoJSON

```
{
 "geometry":{
  "coordinates":[151.17762756347, -33.92598342895],
  "type":"Point"
 },
 "type":"Feature",
 "properties":{
  "Speed":"19.2",
  "TripStartDate":"20200121",
  "TripScheduleRelationship":"SCHEDULED",
  "OccupancyStatus":"MANY_SEATS_AVAILABLE",
  "TripStartTime":"02:00:00",
  "RouteID":"2437_N20",
  "Timestamp":"1579530867",
  "TripID":"883447",
  "Bearing":"77.0",
 }
}
```

For executing spatial joins, we fetched the *planet* data set from the *OpenStreetMap Project* [18], converted this data set into GeoJSON, and stored it in the BBoxDB cluster. The dataset contains the geographical information of the whole world. We imported the *roads* (146 060 493 elements - 67 GB) and the *forests* (5 187 592 elements - 5.4 GB) in our BBoxDB cluster for the demonstration.

3.2 The GUI of BBoxDB

The GUI of BBoxDB shows information about the BBoxDB cluster, the data distribution, and can be used to perform queries. BBoxDB Streams enhances the GUI in such a way that continuous queries are supported. The GUI is optimized for the handling of GeoJSON encoded data. On the main screen, a map of the world (fetched dynamically from the OpenStreetMap Project) is shown. The mouse can be used to create a query rectangle, and a window that is automatically opened allows one to specify the desired query. Continuous range and continuous spatial joins can be executed, and user-defined filters can be applied.

The geometries of the result tuples are shown as an overlay over the map. In addition to the location, the stream elements contain further information. Placing the mouse cursor over an element opens a tooltip. The tooltip contains all the additional information that is contained in the GeoJSON object (e.g., the height of an aircraft or the trip id of a bus). The area of the GUI below the map shows details about the used cluster (i.e., IP, software version, available disks, disk space, CPUs).

Different practical queries can be formulated and observed in the GUI. Figure 3 shows the visualization of an ADS-B data stream of aircraft in the region of Berlin, Germany. Another example (see Figure 4) shows the visualization of a GTFS data stream of metro buses in Sydney, Australia. Additional operations, such as spatial joins between forests and buses (*Which bus drives currently through a forest*?) or roads and buses (*Which buses are driving on the Elizabeth Street in Sydney?*) can be performed and displayed in the GUI.

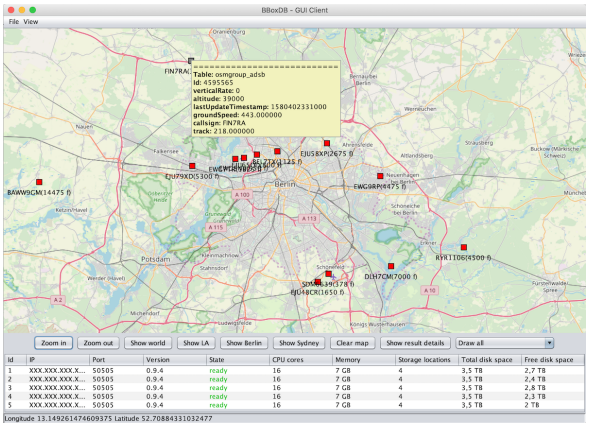


Figure 3: Observing aircraft traffic over Berlin, Germany.

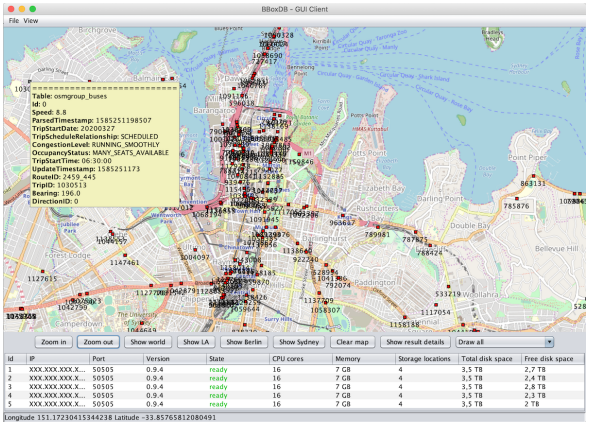


Figure 4: Observing bus traffic in Sydney, Australia.

4 RELATED WORK

BBoxDB and BBoxDB streams have related work in the area of key-value stores and stream processing systems.

Key-Value Stores: During the last decade, NoSQL databases have become popular. They omit features from RDBMS, such as transactions and permanent consistency. This allows NoSQL systems to scale better horizontally. Distributed key-value stores such as *Cassandra* [9] or *HBase* [1] provide simple methods to manage large amounts of key-value pairs. These are optimized for one-dimensional data, since handling n-dimensional data is a laborious task in such systems (see Section 2).

KVS with support for n-dimensional data: *MD-HBase* [16] is a multi-dimensional extension of HBase that allows the efficient storage and retrieval of multi-dimensional data. MD-HBase employs *Quad-Trees* and *K-D Trees* together with a Z-Curve to build an index. Systems such as *EDMI - Efficient Distributed Multi-dimensional Index* [21], *Pyro* [10], and *HGrid* [8] are also enhancements of HBase which use an additional index layer to store multi-dimensional data in HBase. However, operations such as spatial joins or continuous queries are not supported by these systems, and these systems only support point data.

Stream Processing Systems: *Apache Flink* [6], *Apache Spark Streams* [11], *Apache Storm* [2], and *Apache Kafka* [12] are widespread stream processing systems. These systems are not optimized to compare the stream data with larger previously-stored datasets. Operations like geometric indexing or spatial joins are not supported by these systems. In [20] an extension of Apache Storm for the handling of spatial data streams is proposed. However, the paper focuses only on 2-dimensional point data; BBoxDB streams can handle n-dimensional point and non-point data.

5 CONCLUSION

In this demonstration, we have shown the capabilities of BBoxDB Streams for the first time. Two real-world data streams are captured and processed. Queries such as continuous range queries and continuous spatial joins are performed on these streams. The spatial joins are executed between the dynamic data from the data stream and static stored data fetched from the OpenStreetMap Project. The results of the queries are visualized using an enhanced version of the GUI of BBoxDB and user-defined filters are used to refine the bounding box based operations of the query processor. In this demonstration, many aspects of the architecture of the system are only superficially addressed. Topics such as the integration with BBoxDB, the scalability, the filters, and transformations have to be described more precisely. We plan to discuss these topics in detail together with an experimental evaluation of BBoxDB Streams in a full research paper.

REFERENCES

[1] Apache HBase 2020. Website of Apache HBase. https://hbase.apache.org/ [Online; accessed 03-Nov-2020].

[2] Apache Storm 2021. Website of the Apache Storm project. https://storm.apache.org/ - [Online; accessed 20-Jan-2021].

[3] BBoxDB 2018. Website of the BBoxDB project. http://bboxdb.org [Online; accessed 03-Nov-2020].

[4] J. L. Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517.

[5] R. A. Finkel and J. L. Bentley. 1974. Quad Trees a Data Structure for Retrieval on Composite Keys. *Acta Inf.* 4, 1 (March 1974), 1–9.

[6] E. Friedman and K. Tzoumas. 2016. *Introduction to Apache Flink: Stream Processing for Real Time and Beyond* (1st ed.). O'Reilly Media, Inc.

[7] A. Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD Rec.* 14, 2 (June 1984), 47–57.

[8] D. Han and E. Stroulia. 2013. HGrid: A Data Model for Large Geospatial Data Sets in HBase. 910–917.

[9] A. Lakshman and P. Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40.

[10] S. Li, S. Hu, R.K. Ganti, M. Srivatsa, and T.F. Abdelzaher. 2015. Pyro: A Spatial-Temporal Big-Data Storage System. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA.* 97–109.

[11] Z. Nabi. 2016. *Pro Spark Streaming: The Zen of Real-Time Analytics Using Apache Spark* (1st ed.). Apress, Berkely, CA, USA.

[12] N. Narkhede, G. Shapira, and T. Palino. 2017. *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale* (1st ed.). O'Reilly Media, Inc.

[13] J.K. Nidzwetzki and R.H. Güting. 2018. BBoxDB - A Scalable Data Store for Multi-Dimensional Big Data (Demo-Paper). In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management* (Torino, Italy) *(CIKM '18)*. ACM, 1867–1870.

[14] J.K. Nidzwetzki and R.H. Güting. 2019. Demo Paper: Large Scale Spatial Data Processing With User Defined Filters In BBoxDB. In *2019 IEEE International Conference on Big Data (Big Data)*. 4125–4128.

[15] J.K. Nidzwetzki and R.H. Güting. 2020. BBoxDB: A Distributed and Highly Available Key-Bounding-Box-Value Store. *Distributed and Parallel Databases* 38 (June 2020), 439—-493.

[16] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. 2011. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *Proceedings of the 2011 IEEE 12th International Conference on Mobile Data Management - Volume 01 (MDM '11)*. IEEE Computer Society, 7–16.

[17] Open Data Hub 2020. The Open Data Hub for New South Wales transport data. https://opendata.transport.nsw.gov.au - [Online; accessed 03-Nov-2020].

[18] OpenStreetMap Project 2020. Website of the OpenStreetMap Project. http://www.openstreetmap.org - [Online; accessed 03-Nov-2018].

[19] Website of adsbhub.org 2020. The Website of the adsbhub.org project. http://adsbhub.org - [Online; accessed 03-Nov-2020].

[20] F. Zhang, Y. Zheng, D. Xu, Z. Du, Y. Wang, R. Liu, and X. Ye. 2016. Real-Time Spatial Queries for Moving Objects Using Storm Topology. *ISPRS International Journal of Geo-Information* 5, 10 (2016).

[21] X. Zhou, X. Zhang, Y. Wang, R. Li, and S. Wang. 2013. Efficient Distributed Multi-dimensional Index for Big Data Management. In *Proceedings of the 14th International Conference on Web-Age Information Management* (Beidaihe, China) *(WAIM'13)*. Springer-Verlag, Berlin, Heidelberg, 130–141.

Correlation graph analytics for stock time series data

Tong Liu, Paolo Coletti, Anton Dignös, Johann Gamper and Maurizio Murgia
Free University of Bozen/Bolzano, Bozen/Bolzano, Italy
{name.surname}@unibz.it

ABSTRACT

Stock market events are hard to model. In recent years, one approach that has been receiving increasing attention is to analyze graphs induced by price correlations of different stock companies. By analyzing the structure of such graphs, it is possible to identify critical events, e.g., market crises. To the best of our knowledge, there are no tools available that offer comprehensive support for such analyses. This paper introduces a novel tool that offers in-depth analysis with the ability of fine tuning parameters with an intuitive user interface. With a proposed workflow to handle time series data, the tool becomes versatile and it can analyze correlation graphs of different semantics: minimum spanning tree, graphs with edge thresholds, and evolving graphs. It also provides a rich set of functions that enable users to explore easily, interactively and systematically the correlation graphs starting from a file of raw time series data. With real-world stock data, we demonstrate how straightforward yet effective it is to accomplish various analytical tasks with the proposed tool.

1 INTRODUCTION

Time series systems are pervasively used in many domains for different tasks such as monitoring and recording data gathered over time, with which analysts can discover meaningful and valuable information to understand the observed system and to avoid risks. The examples of applying time-series technologies to real-world problems are numerous, e.g., fault detection [18], seasonal trend analysis [21], financial data [11], etc.

For the analysis of financial data, Mantegna [14] proposed to compute all correlations between the time signals and to visualize them in a graph. This is illustrated in Fig. 1. First, the pairwise correlations between the five input signals are computed and stored in a matrix. Then, the correlation matrix is transformed into a graph, where nodes represent signals and edges are labeled with the correlation coefficient between the two nodes. Instead of visualizing a complete graph, the minimum spanning tree (MST) is often used as a compact overview of the signal correlations.

In recent years, a significant number of works have shown that by observing changes in the structure of a MST, it is possible to deduce important events in stock markets, such as market crises and volatility [5, 6, 16]. Many studies about stock market analysis adopted this approach and investigated the structure and topologies of the induced network or MST [4, 6, 9, 19]. The study of correlation graph analysis also raised the attention of the computer science community. Marti et al. [15] studied the best window length to calculate time series correlations. Azzalini et al. [2] proposed a technique to detect significant changes in financial time series clusters expressed with hierarchical correlation trees. Luo et al. [12] used correlation graphs to spot cheating behaviors with business data. In the database community, research focused on efficient methods for correlation graph

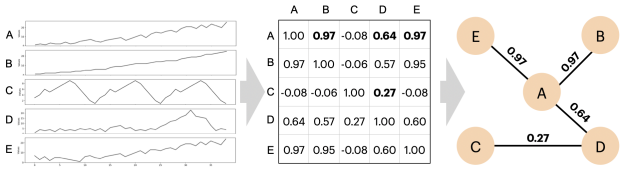


Figure 1: Correlation analysis of time series using MST.

analyses. Petrov et al. [17] proposed different approaches for exploring correlated time signals interactively. Aghasadeghi et al. [1] presented techniques to visualize evolving graphs at different resolutions. Likewise, Mamun et al. [13] studied efficient solutions for computing MSTs.

When it comes to correlation graph analytics for financial data, analysts have to tackle several challenges: identifying appropriate parameter values such that the computed correlation graph is meaningful and presents visible characteristics; investigating whether patterns exist for graphs under different representational semantics, such as MST, graphs with specific thresholds, graphs of signal classes; assessing the robustness of graph features (e.g., radius, degree distribution), and determining if they are reliable to capture special events. Existing tools in the finance literature [4, 6, 9, 19] suffer at least from the following two limitations: (i) they offer only a few of the above mentioned aspects and (ii) they were implemented ad hoc for specific case studies and are not designed to be easily integrated with other tools or analyses.

In this demo paper, we present a versatile and user-friendly tool for analyzing time series data based on the pairwise correlations that are visualized in a graph. We propose a workflow in which the parameters are considered logically, and it allows a high level of flexibility to visualize and analyze the graphs: the comparison of two graphs representing subsequences of the data selected by two different windows; the evolution and changes of a graph over time by applying a moving window; and a compact heatmap representation of crucial graph parameters for all possible windows over the data. Throughout the paper, we use the financial domain to illustrate the key features of the system. However, the analysis tool can also be applied in other domains, as we will illustrate in one of the demonstration scenarios. The tool is online at https://dbs.inf.unibz.it/projects/ismard/, and people can use it with their data at hand. In four demo scenarios we discuss how to use the tool and what can be learned by a data analyst.

2 SYSTEM OVERVIEW

2.1 Architecture

The system is implemented as a web App based on a client-server architecture (cf. Fig. 2). On the client side, a user can upload time series data, set parameters for data pre-processing and graph computations, and inspect the resulting graphs. By tuning the parameters in the client side, the web page will perform AJAX calls and trigger the processing of the data in the server. The computed graph data is then returned to the client and interpreted by visualization components. On the client side, two Javascript

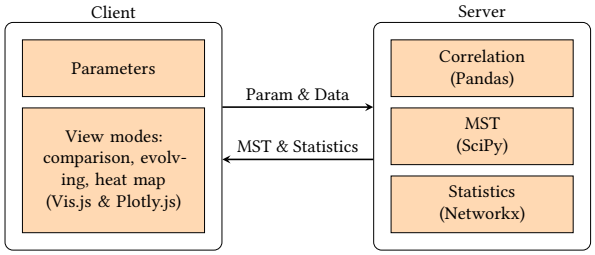


Figure 2: System Architecture

libraries are used for the visualization: VIS.js for graph views and Plotly.js for data reports. On the server side, three Python libraries are employed: Pandas for correlation computation, SciPy for MST computation, and Networkx for graph statistics.

2.2 Data and Parameters

The system accepts time series data in CSV format. The first row specifies the signal names. Each subsequent row stores a time point followed by the signal values for that time point. Missing values are marked with `NaN`. Optionally, a second CSV file containing ⟨signal name, class name⟩-pairs can be uploaded in order to group the time signals into classes.

There are a number of parameters that can be controlled by the user and make the tool flexible for data analytics. For the preparation of the input data, the user can, among others, specify a time granularity (e.g., daily, weekly, or monthly) for the analysis and whether to use the raw data values or the variation of the value w.r.t. the previous time point. The computation of the correlations and the construction of the MST/graph can be controlled by several parameters, including a time range for the analysis, the required minimum number of values for each signal for a meaningful analysis, a minimum overlap between two signals, and a correlation threshold for an edge to be included in the graph visualization. Finally, for the visualization of the analysis results three different views are offered.

2.3 Analysis Workflow

The first step in the analysis workflow is to load the raw data file, store it on the server, and then to optionally apply two *preprocessing* operations. First, the user can specify a time granularity for the analysis which is different from the granularity of the raw data. For instance, in the financial sector the raw data typically contain the daily closing prices, but financial analysts often prefer to use weekly or monthly price values in order to reduce noise. Instead of computing the average, the closing price of Thursday is used as the weekly price, and the price of the last trading day of each month as the monthly price. To support the analysis of data from other domains, the tool also offers the use of the average value in combination with different granularities. Second, the analysis of stock data usually investigates the variation of the price rather than the price itself. Two different variations are frequently used: the return rate, which is the relative price difference to the previous time point, i.e., $r_i = (p_i - p_{i-1})/p_{i-1}$ with p_i being the closing price at time point t_i; and the log return $r_i = (\log p_i - \log p_{i-1})$. The result of the pre-processing phase is stored in the so-called working file on the server.

The next step is to compute the *pairwise correlations* matrix using the Pearson Correlation coefficient. For this, the subsequences of the signals that correspond to the specified window are loaded from the working file into a so-called DataFrame — an efficient two-dimensional main memory data structure of the Pandas library, which natively supports the computation of the Pearson correlation. To obtain robust and reliable results, time series that contain an excessive number of missing values are omitted from the computation of the correlation. Similarly, we do not compute the correlation between two time series if they do not sufficiently overlap in time. Both parameters can be controlled by the user. For the time series that pass these filters, the Pearson Correlation coefficient is computed, which for two time signals x and y with average value $\bar{x}$ and $\bar{y}$, respectively, is defined as $r = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i-\bar{y})^2}}$ with $-1 \leq r \leq 1$. A value of 1 indicates total positive linear correlation, a value of -1 total negative linear correlation, and a value of 0 no linear correlation. Instead of analyzing the correlation between individual stock signals, the user can decide to investigate the stock sectors (classes), which allows them to gain insights into entire sectors rather than individual stocks. The signal of a class is computed as the average of all stock signals belonging to that class. More advanced aggregation methods are known in the literature, e.g., the free-float adjusted market-capitalization weighting method [10], which computes a weighted average based on the stock shares.

Next, we compute the *minimum spanning tree (MST)* from the correlation matrix using the classical algorithm by Kruskal [7]. Each node represents a stock signal, and the edges are labeled with the correlation coefficient between the connected nodes. As an alternative to the MST, a correlation graph can be used for the visualization. Since a complete graph would not be very useful for analysis purposes, the user can prune weak edges by introducing a threshold for the correlation.

The last step is to compute several *statistics* for the MST/graph. A very important measure for the analysis is the degree distribution of the graph nodes. This measure reflects structural information of a graph and can be used to detect stock market events [3, 6]. Other useful statistics include the range of the actual correlation values and the radius of the MST/graph. Additionally, we also compute the modularity score [8, 20], which allows us to measure whether the stock connections in a MST correlate with the stock sectors. This score is defined as $\sum_{i=1}^{k}(e_{ii} - a_i^2)$, where k is the number of sectors, e_{ii} is the percentage of edges connecting two nodes of the same sector i, and a_i is the percentage of edges for which at least one node belongs to sector i.

2.4 Interaction and Result Visualization

For the visualization of the analysis results, our tool offers three principal viewing modes: comparison mode, evolving graphs mode, and heatmap mode.

The goal of the *comparison mode* is to analyze the data over two different time periods in order to spot similarities and differences, e.g., comparing a normal period with a crisis period. Figure 3 shows a screenshot of this mode, which contains two panels for the two graphs. A scroll bar allows users to select a time window for the analysis. The graphs report the node names (either signal name or class name), and edges are labeled with the correlation coefficients. By clicking on a node, additional information is shown such as the node's eccentricity and class. A color coding is used for the graphs. The node color indicates the class of the represented stock. For the edges, the colors distinguish different levels of correlation: red for high correlations above 0.7, orange for medium correlations between 0.4 and 0.7, and gray for low correlations below 0.4. At the bottom of the screen, the computed statistics are summarized.

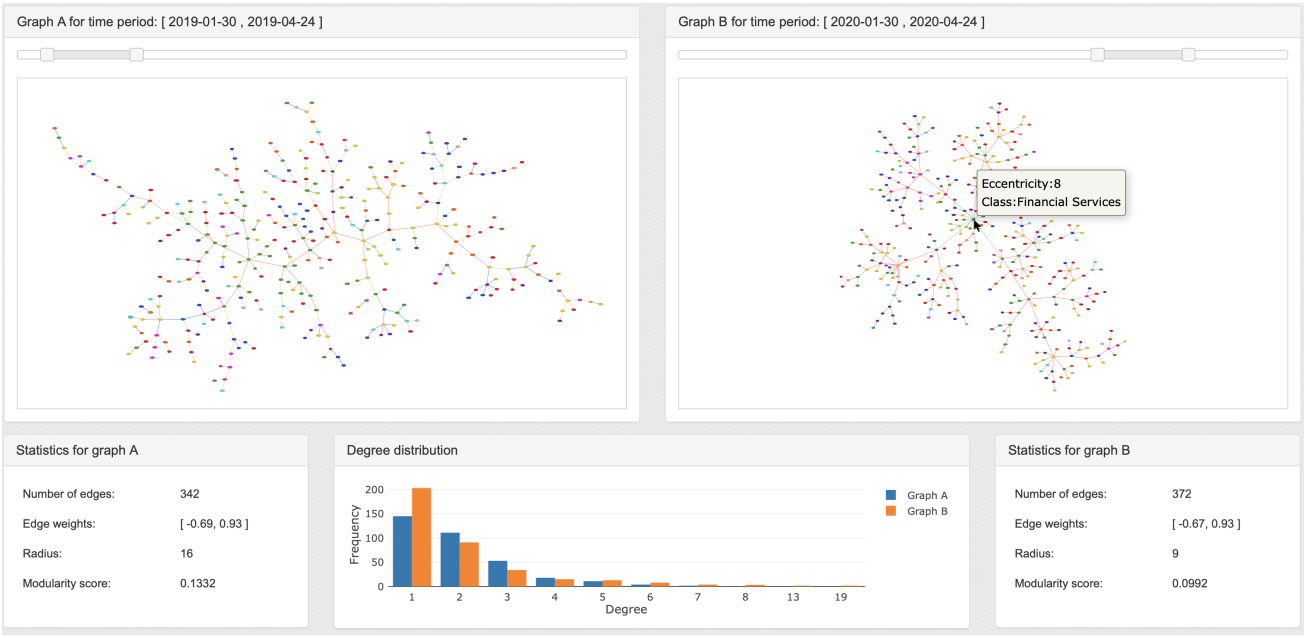


Figure 3: Compare MSTs in ordinary period with COVID crisis period.

The *evolving graph* mode shows the evolution of a graph along the time axis, which helps to understand the subtle changes of graph components over time. For this, a sliding window is used, where the user can specify the window length and the step size. For each window a graph is computed. To highlight the changes between consecutive windows, the new edges w.r.t. the previous graph are shown by dashed lines (cf. Fig. 4).

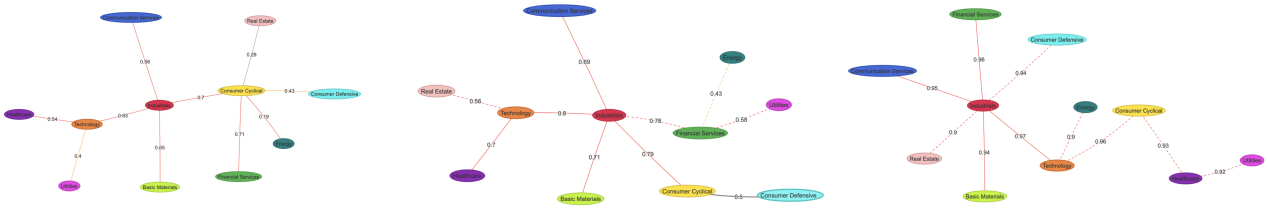
The *heatmap* mode helps to understand the stability and reliability of a MST metric w.r.t. the values of the selected time window. For this, we compute a heatmap for three important graph metrics: first degree score defined as the number of nodes with degree one divided by the total number of nodes; the modularity score; and the radius. Each heatmap shows the metric for all possible windows over the data (cf. Fig. 5). The x-axis represents all possible starting points of time windows, and the y-axis all possible window lengths. Heatmaps allow users to investigate whether the metric value changes drastically with different time windows, as well as discern whether a metric's values are similar with time windows that cover special events in the timeline. The heatmaps are interactive. By clicking on the heatmap, the system computes the MST over the corresponding window. This allows us to inspect the MST in detail. Since the computation of the heatmaps in real time is a computational bottleneck, this feature is currently only activated for the demo dataset. An efficient computation of the heatmaps for large datasets is part of our future work.

3 DEMONSTRATION SCENARIOS

The following four scenarios provide a glimpse of how our tool can be used to systematically investigate behaviors and properties of time series data. We use the Italian stock market data collected from Yahoo Finance. This dataset contains the daily stock prices of 407 companies which traded from January 2019 to June 2020 and includes the COVID stock crash. The companies are classified into 12 market sectors, such as financial service, real estate, energy, etc. In addition, we use a dataset that contains signals from industrial devices provided by a local company.

Comparing Normal and Crisis Periods. In the first demo scenario, an analyst is interested in investigating how stock correlations were affected by the COVID crash. This can be observed by graph shapes and indicators in the comparison mode shown in Fig. 3. The left-hand side shows the MST over a normal period (Jan 2019 – Apr 2019), while the right-hand side shows the MST during the COVID crisis (Jan 2020 – Apr 2020). From the graph shapes, analysts can observe that the crisis led to the formation of many star-like hubs and red-color edges, indicating that a large number of companies were strongly correlated. From the indicators, analysts find that during the COVID crisis the MST had a much smaller radius and a steeper degree distribution compared to the normal period. In particular, significantly more nodes had a degree of 1 during the crisis period. Similar analyses can be performed for other market events. More details on shapes and indicators are described in [6]. Moreover, different parameter values and time windows can be chosen to examine the stability of the above features.

Changes over Time. In this scenario, an analyst wants to investigate whether the correlation between different sectors has changed in proximity to the COVID crash. This type of analysis is supported in the evolution mode. Figure 4 shows the MST computed over three consecutive windows over the aggregated stock signals. The length of the sliding window is three months, and the step size one month. A dashed line indicates a new edge compared to the MST over the previous window. It can be observed that the Industrials sector (red node) tends to be at the center of the tree. Presumably, sectors in Fig. 4c that are connected with Industrials are the most negatively affected sectors by the COVID crash. In contrast, Healthcare (purple) switched its connection from Technology to Consumer Cyclical and Utilities. They could be the positively influenced sectors, since with the medical research and "stay at home" order, these sectors either received more investments or increased consumption.



(a) 10/2019–01/2020 (b) 11/2019–02/2020 (c) 12/2019–03/2020

Figure 4: Evolving graphs.

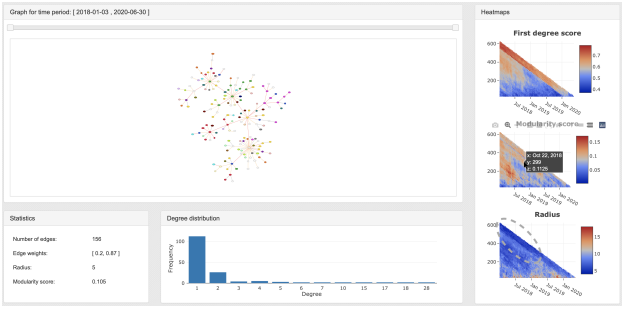


Figure 5: Heatmap view.

Concise Overview of All Windows. A critical aspect for time series analysis is to choose an appropriate window length, owing to the fact that real-world data are often noisy and contain missing values. The heatmap mode helps in this regard as it provides a concise overview of important MST metrics over all possible windows. An example is shown in Fig. 5 with three heatmaps for the first degree score, the modularity score, and the radius of the MST, respectively. It can be observed that for long time windows (i.e., more than 300 time points) the radius is more reliable to signal the crisis period. This is explained by a dark strip at the top border of the heatmap (gray circle in Fig. 5). It appears that for all long time windows that cover the crisis period, the radius of the corresponding MST is significantly lower than the MSTs in normal periods. A similar pattern can be observed for the heatmap of the first degree score. In this scenario, a user learns which are sound indicators for detecting crises and the conditions to use them, e.g., the appropriate window length.

Beyond Stock Data. To show the generality of our tool, we also provide a demonstration scenario for the analysis of data from the industrial sector. The time series come from sensors monitoring components of large industrial printers, such as the temperature of internal CPUs, ink speed, or belt velocity. In this scenario, we show how our tool can be used to understand the interaction of different components, similar to stock sectors, based on their correlation over different time periods.

4 CONCLUSIONS AND FUTURE WORK

In this demo paper, we presented a new web App to investigate stock time series data. The key idea is to compute the pairwise correlations between the data and – in order to facilitate the analysis – to visualize them in a minimum spanning tree, where nodes represent the trading companies and edges show their correlation. With Italian stock market data we demonstrated the effectiveness and versatility of our tool.

Future work points in two directions. First, we will investigate more efficient algorithms for the computation of heatmaps in order to make the tool scalable for larger datasets. The other direction concerns the use of machine learning techniques to detect useful parameter configurations for the analysis automatically.

ACKNOWLEDGMENTS

This work is supported by the projects ISMarD and TASMA, which are funded by the Free University of Bozen-Bolzano.

REFERENCES

[1] Amir Aghasadeghi, Vera Zaychik Moffitt, Sebastian Schelter, and Julia Stoyanovich. 2020. Zooming Out on an Evolving Graph. In *EDBT 2020*. 25–36.
[2] Davide Azzalini, Fabio Azzalini, Mirjana Mazuran, and Letizia Tanca. 2019. Tracking the Evolution of Financial Time Series Clusters. In *DSMM@SIGMOD 2019*. 4:1–4:5.
[3] AQ Barbi and GA Prataviera. 2019. Nonlinear dependencies on Brazilian equity network from mutual information minimum spanning trees. *Physica A: Statistical Mechanics and its Applications* 523 (2019), 876–885.
[4] Xuewei Cao, Yongbin Shi, Penghao Wang, Liujun Chen, and Yougui Wang. 2018. The evolution of network topology structure of Chinese stock market. In *ICBDA 2018*. IEEE, 329–333.
[5] Ricardo Coelho, Claire G Gilmore, Brian Lucey, Peter Richmond, and Stefan Hutzler. 2007. The evolution of interdependence in world equity markets-Evidence from minimum spanning trees. *Physica A: Statistical Mechanics and its Applications* 376 (2007), 455–466.
[6] Paolo Coletti and Maurizio Murgia. 2016. The network of the Italian stock market during the 2008–2011 financial crises. *Algorithmic Finance* 5, 3-4 (2016), 111–137.
[7] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.
[8] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.
[9] Raphael H Heiberger. 2014. Stock network stability in times of crisis. *Physica A: Statistical Mechanics and its Applications* 393 (2014), 376–381.
[10] S&P Dow Jones. 2014. Index mathematics methodology.
[11] Kyoung-jae Kim. 2003. Financial time series forecasting using support vector machines. *Neurocomputing* 55, 1-2 (2003), 307–319.
[12] Ping Luo, Kai Shu, Junjie Wu, Li Wan, and Yong Tan. 2020. Exploring Correlation Network for Cheating Detection. *ACM Trans. Intell. Syst. Technol.* 11, 1 (2020), 12:1–12:23.
[13] Abdullah-Al Mamun and Sanguthevar Rajasekaran. 2016. An efficient Minimum Spanning Tree algorithm. In *ISCC 2016*. 1047–1052.
[14] Rosario N Mantegna. 1999. Hierarchical structure in financial markets. *Eur. Phys. J. B* 11, 1 (1999), 193–197.
[15] Gautier Marti, Sébastien Andler, Frank Nielsen, and Philippe Donnat. 2016. Clustering Financial Time Series: How Long Is Enough?. In *IJCAI 2016*. 2583–2589.
[16] Salvatore Micciché, Giovanni Bonanno, Fabrizio Lillo, and Rosario N Mantegna. 2003. Degree stability of a minimum spanning tree of price return and volatility. *Physica A: Statistical Mechanics and its Applications* 324, 1-2 (2003), 66–73.
[17] Daniel Petrov, Rakan Alseghayer, Mohamed A. Sharaf, Panos K. Chrysanthis, and Alexandros Labrinidis. 2017. Interactive Exploration of Correlated Time Series. In *ExploreDB 2017*. 2:1–2:6.
[18] Francisco Serdio, Edwin Lughofer, Kurt Pichler, Thomas Buchegger, Markus Pichler, and Hajrudin Efendic. 2014. Fault detection in multi-sensor networks based on multivariate time-series models and orthogonal transformations. *Information Fusion* 20 (2014), 272–291.
[19] M Wiliński, A Sienkiewicz, Tomasz Gubiec, R Kutner, and ZR Struzik. 2013. Structural and topological phase transitions on the German Stock Exchange. *Physica A: Statistical Mechanics and its Applications* 392, 23 (2013), 5963–5973.
[20] Lisi Xia, Daming You, Xin Jiang, and Quantong Guo. 2018. Comparison between global financial crisis and local stock disaster on top of Chinese stock network. *Physica A: Statistical Mechanics and its Applications* 490 (2018), 222–230.
[21] G Peter Zhang and Min Qi. 2005. Neural network forecasting for seasonal and trend time series. *Eur. J. Oper. Res.* 160, 2 (2005), 501–514.

Conquering a Panda's weaker self - Fighting laziness with laziness

Demo Paper

Stefan Hagedorn
TU Ilmenau
Ilmenau, Germany
stefan.hagedorn@tu-ilmenau.de

Steffen Kläbe
TU Ilmenau
Ilmenau, Germany
steffen.klaebe@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau
Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

The Python programming language has become very popular among data scientists because of its easy-to-learn syntax and rich ecosystem of libraries. Especially the Pandas framework is widely used for various data processing and analytics tasks. However, due to its memory management and eager evaluation Pandas does not scale and workstations quickly come to their limits even for moderate data set sizes. With Grizzly, we introduce a framework that produces SQL queries for operations on `DataFrames`, moving complexity from workstations to database servers. Grizzly allows to not only access data already stored in a database, but also to combine it with external data from files. Furthermore, users can use their own user-defined functions or use Grizzly's model join feature to easily apply machine learning models to data, both being executed inside the database server. This allows for fast and scalable data analytics operations, even with a small workstation.

1 INTRODUCTION

Data Science and Machine Learning are hot topics, not only in research but also in industrial and commercial applications. Although the terms *Data Analysis* and *Data Science* date back to the early 1960s and 1970s, respectively, with the rise of *Big Data* in the early 2000s more and more companies started to collect and analyze every piece of information they could generate about their, e.g., sales and customers with the goal to gain insights that help to improve the companies productivity and business. One of the standard languages for data science tasks is Python (besides Julia and R). Python has become very popular because of its easy to learn syntax that allows to quickly build prototypical data processing pipelines. One of the most popular libraries for Python in the field of data analytics is Pandas. It allows to easily load data from various sources and represents it in `DataFrames`– a table-like abstraction with column names, types and more meta information. Using Pandas, one can load data in various formats from local or remote locations into `DataFrames` and apply operations such as projection, filter, grouping, join – all well known from the relational algebra. In Pandas, these operations are executed on the local machine of the data scientist and create copies of the data in the local RAM. Since this is slow and means larger-than-RAM data sets cannot be processed easily, data scientists who should actually focus on their data analytics tasks, started to build their custom solutions for parallel processing or buffer management. However, the database community has built fast, robust, and scalable systems to perform exactly this kind of operations efficiently, even if the complete data set does not fit into RAM. Often the data to analyze is already stored in such reliable database systems. Thus, instead of moving the data from the large DBMS server into the potentially small data scientist's laptop for processing, the operations defined in the Python script should be transferred to the DBMS and be translated into a language it understands, i.e. SQL.

During recent years, a few systems have been developed to address these needs, such as RIOT-DB [12], AIDA [3], Modin [9], AFrame [10], and IBIS[1]. IBIS was initiated by the creators of Pandas and tries to overcome the scalability issues of Pandas by using lazy evaluation and converting operations on `DataFrames` into a (sequence of) SQL queries. However, it is not possible to join data from different database systems and UDFs can only be executed using the Pandas backend (i.e. local execution) or on Google Big Query. The `DataFrame` concept has also been adopted by other frameworks like Apache Spark [11], Koalas[2], and Nvidia Rapids[3]. Internally, these systems use optimizers to tune the query and produce good execution plans. The idea of providing a `DataFrame` API over graph data has been studied for example in [2] and [8].

In this paper, we demonstrate our Grizzly[4] framework for transpiling Python code to SQL queries, with the following features:

- Grizzly uses query-shipping and lazy evaluation to achieve high scalability.
- It supports relational operations in standard SQL syntax and uses configurable templates for DBMS-specific dialects without changing the underlying execution system.
- External data can be combined with in-DBMS data.
- Users can define their own functions (UDFs) in Python to apply within the generated query.
- Grizzly uses the UDF mechanism to load and execute pre-trained machine learning models inside the database.

In our demonstration, we show how easy it is to exchange Pandas with Grizzly as the execution engine for `DataFrame` programs. Using a web application, users can compare Pandas and Grizzly scripts, either using our provided examples for various scenarios or writing own code, side-by-side and run them on prepared datasets. Our demo system automatically generates comparison charts for query performance and memory consumption. As these charts are maintained over multiple runs, users can built their own evaluations by varying parameters like the dataset size.

2 REQUIREMENTS

Companies typically store their valuable data in some durable and integer database. This database consists of various tables,

[1] http://ibis-project.org/
[2] https://www.github.com/databricks/koalas
[3] https://developer.nvidia.com/rapids
[4] https://github.com/dbis-ilm/grizzly

containing, e.g., the customer information, the products the company sells, and of course the orders the customers have made over time. This basically resembles the well-known TPC-H [1] benchmark and for larger companies this database can quickly store several GBs. Below we briefly describe use case scenarios from which the requirements for a system that shifts the processing into the DBMS can be derived.

Basic Data Processing. As an example use case, a data scientist may be interested in the names of the customers, who ordered a specific product most often. Using Pandas, the scientist would load the tables `customer`, `orders`, and `products` onto her local workstation and then filter the products by the product name she is interested in, join the remaining products with orders and then with customers. Finally, the join result may be grouped by the customer and aggregated to count how often they have ordered the specific product.

At this point we would like to emphasize that with Pandas, all these processing steps take place in the scientists workstation and not on the actual database server that stores this data. Though, Pandas is able to send a SQL query to a database server in order to fetch proprocessed data. However, we argue that many data scientists often don't know or use SQL and prefer their higher level languages. Thus, to improve the query performance and often make the evaluation possible at all, the computational part should be shifted into the database server while letting the users still express their analysis tasks in Python.

Accessing External Data. After getting these top customers, the data scientist needs to find out how these products are perceived in social media platforms. The company may track social media posts using some additional systems and collects the product reviews in a text file. However, these posts are not critical to the company and are not ingested into the DB. Thus, the data scientist needs to join the data in the DB with the data in files. Currently, this operation would typically also be executed on the workstation, but should optimally be executed by the DBMS. In order to do so, the DBMS must import the file as a (temporary) table. This import should be transparent to the user so that she only needs to specify a file path, but does not have to bother with DBMS specific import code.

UDFs & Model Join. The well-known data processing libraries all provide a vast variety of algorithm implementations and operators for various tasks. However, these may not be enough for every problem to solve and users fall back to implement their own logic in user-defined functions which they want to apply. Such functions could either be applied to every record individually or to one table/data set as a whole. Again, the function could be applied on the workstation using Pandas, but then subsequent operations which could be handled in the DBMS are not possible anymore. Thus, the function's code should also be shipped to the database server transparently for convenient execution.

A special case is the application of some existing machine learning model. An existing model like RoBERTa [6] may be trained to detect the sentiment of some text, i.e. if it is positive or negative. After the data scientist joined the top customers and products with the product reviews from a text file, she needs to classify how the reviews rate the corresponding product. Thus, she needs to join the model with the data (intermediate query result) in the database. For this, it must be possible to select and load an existing model and join it with the data in the database, i.e. apply it to every tuple of a table/intermediate result. In further discussion we name this concept model join.

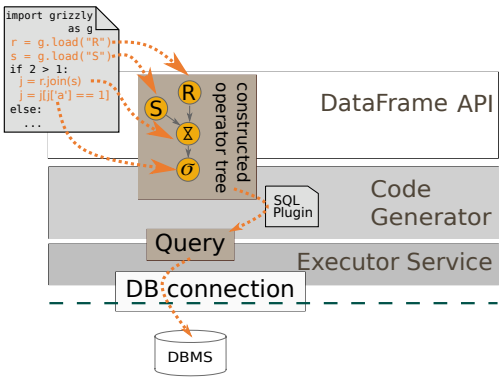
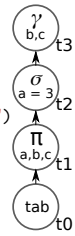


Figure 1: Overview of Grizzly's architecture.

```
# load table (t0)
df = grizzly.read_table("tab")
# projection to a,b,c (t1)
df = df[['a','b','c']]
# selection (t2)
df = df[df.a == 3]
# group by b,c (t3)
df = df.groupby(['b','c'])
```

(a) Source Python code.



(b) Operator tree.

```
SELECT t3.b, t3.c FROM (
  SELECT * FROM (
    SELECT t1.a, t1.b, t1.c
    FROM (
      SELECT * FROM tab t0
    ) t1
  ) t2 WHERE t2.a = 3
) t3 GROUP BY t3.b, t3.c
```

(c) Produced SQL query.

Figure 2: Steps for transpiling Python code to a SQL query: The operations on `DataFrames` (a) are collected in an intermediate operator tree (b) which is traversed to produce a nested SQL query (c).

3 GRIZZLY ARCHITECTURE

The scalability issues of the Pandas framework are a consequence of two major bottlenecks:

- Pandas operations are executed eagerly, producing numerous intermediate results that increase the memory consumption of a Pandas program.
- Pandas uses the data-shipping paradigm, which means that data is transferred to the place where the program is executed. Despite being easy to use, the data-shipping paradigm limits scalability as potentially large amounts of data are transferred. Operators are executed on the client-side, which usually consists of weaker hardware compared to (cloud) servers.

The design of the Grizzly framework focuses on solving these two problems. First, Grizzly replaces the eager operator execution approach with a lazy approach by collecting operators and generating a query when the result is needed. Second, Grizzly combines the convenience of the data-shipping paradigm with the scalability of the query-shipping paradigm by abstracting from data access and pushing query execution to the DBMS.

An overview of the Grizzly architecture is given in Figure 1. Grizzly provides a Pandas-like `DataFrame` API for compatibility with existing Pandas programs. In order to achieve the lazy execution paradigm, Grizzly collects operators and builds a lineage graph as an internal query representation, following a similar approach as RDDs in Apache Spark [11]. Operators are classified as transformations (e.g. projections, filters, joins) or actions (e.g. show, count, sum). While transformations are collected in the lineage graph, actions trigger code generation and execution. In order to meet the second design goal of using the query-shipping

paradigm, Grizzly transpiles the lineage graph of `DataFrame` to standard SQL, compatible with a broad range of DBMSs. The transpilation is achieved by traversing the lineage graph and using a mapping between `DataFrame` operators and SQL query constructs, which is described in detail in [4]. There are two options for incrementally constructing a SQL query:

(1) Incrementally extend `SELECT`, `FROM` and `WHERE` blocks of a (single) SQL query, or
(2) generate a separate query for each operator and nest the existing query in the `FROM` clause.

While producing more compact and easier-to-read queries, option (1) has the drawback that nesting is still required in some cases, e.g. for a filter on a computed column. These cases require special handling in the code generation as well as a careful naming. We observed that modern query optimizers of existing DBMSs have excellent capabilities for unnesting queries, and thus decided to follow option (2) and apply a generic naming schema for sub-queries. Queries are send to the DBMS using a standard connection object as specified by PEP 249[5]. Additionally, Grizzly uses a template file in order to match vendor-specific SQL dialects. An example workflow for transpiling a Pandas program into a nested SQL query is shown in Figure 2.

4 API EXTENSIONS

Modern data analytics tasks not only use traditional operators, but also include the use of external data sources, user-defined functions (UDFs) or machine learning models for prediction or classification as we argued in the use case discussion in Section 2. In order to address these challenges, we extended the `DataFrame` API of Pandas with a set of operators. All additional operators are designed with the goal of hiding complexity and providing an easy-to-use interface for complex operations.

The basic approach of supporting the described features is the generation of additional queries to create functions or define external data sources. Such queries are required to be run before the actual analysis starts. Grizzly maintains a list of pre-queries and whenever the lineage graph traversal reaches an operator that requires a pre-query, it is generated and appended to the list. Finally, all pre-queries are automatically executed before the actual generated query.

External Data Sources. Various database systems offer support for external data sources by creating a table over a file, e.g. using foreign data wrappers in PostgreSQL or external tables in Actian Vector. Similar to ordinary database tables using `read_table`, external tables can be used as leaf nodes in the lineage graph. Grizzly offers the `read_external_table` function for this, which takes the path to the external source as well as the schema as parameters and returns a `DataFrame` for further usage. During code generation, the external table is generated using a pre-query and given a temporary name to be referenced in the actual query.

UDF Support. In Pandas, users can apply custom functions to `DataFrames` using the `apply` function in an elementwise fashion (scalar UDFs) or as a reduce function (table UDFs). Modelling the apply operator as an action, and therefore executing the subquery and applying the UDF at the client side, has the major drawback that further operators also need to be executed at the client side. In order to avoid this, we model UDFs as transformations in Grizzly by exploiting the recent upcome of Python UDF support in database systems. The source code of the UDF is accessed via reflection and transferred to the database system using a UDF creation as a pre-query. The function is created with a generic name and is then applied in the actual query in the projection list. Note that many database systems currently only support scalar UDFs and only offer this feature as a beta version due to security concerns. Consequently, Grizzly is currently also limited to scalar UDFs and requires the vendor-specific activation of the UDF feature in order to support UDFs. As a result of modelling UDFs as transformations, UDF computation can also be pushed to the database, enabling efficient subsequent operations on the UDF result.

```python
# Define conversion functions
def input_to_model(a: str):
        ...
def model_to_output(a) -> str:
        ...

# Use external file
df = grizzly.read_external_table("path/to/file", schema, options)
# Apply onnx classification model using conversion functions
df['class'] = df['text'].apply_model("/path/to/model",
↪  input_to_model, model_to_output)
# Count elements per classification
df = df.groupby('class').count()
# Trigger execution and show result
df.show()
```

Listing 1: Example for external file usage and model join

Machine Learning Model Join. Database systems are a non-optimal environment for training complex machine learning models, as this task is mainly performed on massively parallel engines like GPUs and involve a hybrid workload of intensive computations as well as large updates of e.g. weights in the model. However, there are various pre-trained models available that can be easily used for data analytics, e.g. in the Model Zoo on Github[6]. Applying machine learning models to data is a special case of UDFs and can also be applied to Pandas `DataFrames` using the `apply` function and handcrafted code for model execution. Grizzly offers a family of specialized `apply` functions for the most popular model formats and execution engines PyTorch[7], Tensorflow[8] and ONNX[9]. Similar to UDFs, these operators are modelled as transformations and designed for comfortable usage, demanding only necessary, model type specific parameters from the user. The generated code for model execution (application) is handled like a UDF: it is defined in the database system using a pre-query and used in the projection list of the generated SQL query. For a more detailed discussion of challenges that come with this features and their solutions in Grizzly, we kindly refer to [5].

For the discussion of the presented features we assume that necessary files like data sources or machine learning models are accessible from the database server and that required Python modules are installed. We argue that using cloud file systems or NAS storage this is not a limitation and that root access or contacting the database administrator is currently necessary anyway in order to use the UDF feature.

As an example, Listing 1 shows a program that loads an external file, applies a classification model in the ONNX format on column *text* and then counts the number of entries per class. ONNX models are typically provided with conversion functions to convert an input to a tensor and convert the output tensor

[5]https://www.python.org/dev/peps/pep-0249/

[6]https://www.github.com/onnx/models
[7]https://www.pytorch.org/
[8]https://www.tensorflow.org/
[9]https://www.github.com/onnx/

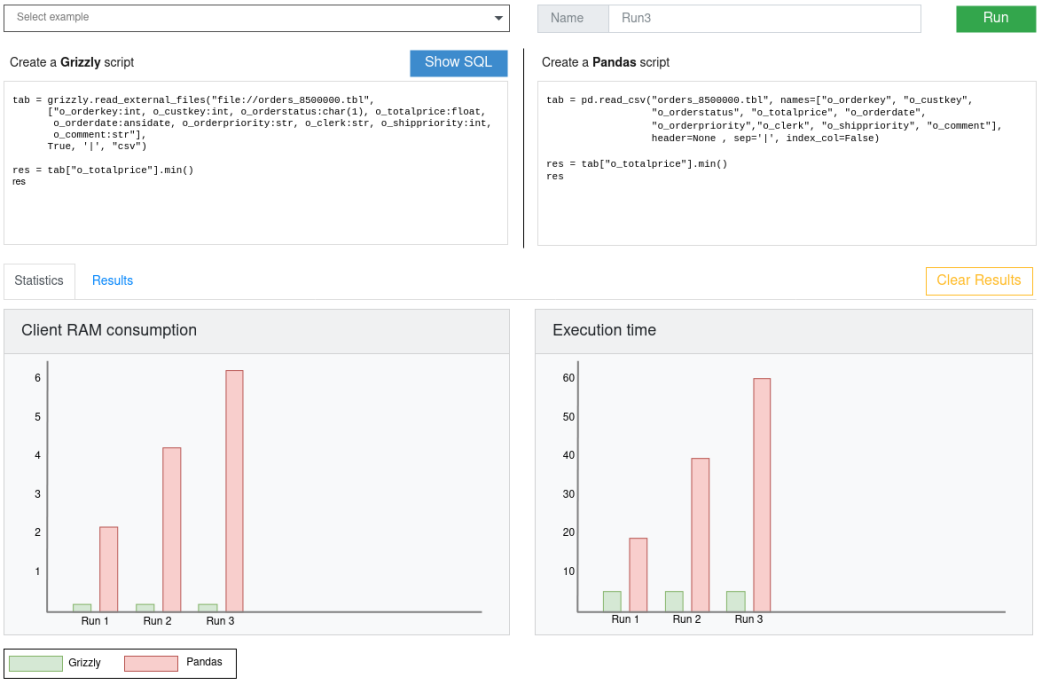


Figure 3: Screenshot of the Grizzly Web Application.

back. The type hints in the signature of these functions are used to determine the type of the overall UDF. The types are required to define the functions in SQL. Grizzly generates pre-queries for the external table and the UDF, ultimately executing three SQL statements.

5 OBSERVING THE BEARS IN THE WILD

The demonstration highlights the benefits users can expect when using Grizzly over Pandas. It invites visitors to interact with the system over a web application and Jupyter notebooks.

The application lets users choose between prepared scenarios, but also allows to run own scripts. For both cases we provide a pre-loaded collection of data sets (from TPC-H [1] and IMDB [7]) with varying sizes. The scenarios range from simple queries to more complex tasks that make use of the features we explained in the previous sections: executing UDFs, combining existing tables with external files, and performing the model join. For every scenario, a prepared Grizzly and Pandas script can be selected and executed. The side-by-side code editors demonstrate how similar the Grizzly and Pandas APIs are, but especially for the model join this will also highlight how much work Grizzly saves developers compared to Pandas. Using the "Show SQL" button, one can inspect the SQL query Grizzly transparently generates for an entered Python program. As an example, the model join scenario offers different models available in ONNX, PyTorch, and Tensorflow format to classify entries in the tables. It follows the use case described in Section 2: We connect to a movie database and join this data with reviews from a text file and classify every movie using a pre-trained model into the categories *positive* and *negative*. The final result is grouped in order to count the number of positive and negative reviews per movie. The Grizzly code is as easy as shown in Listing 1, being significantly smaller than the respective Pandas implementation. Additionally, the application shows the effort to handcraft the SQL code, which is transparently generated by Grizzly. Users can also compare the external table feature of Grizzly to the respective `read_csv` feature of Pandas as sketched in Figure 3 and observe a significant performance gain when using Grizzly. The web application tracks the execution time as well as the memory consumption during the execution of a program over multiple runs and visualizes both metrics for comparison. Through the collected result graphs and by using our input data sets of different sizes, users can build their own evaluation and investigate the scalability of the systems.

A second part of the demonstration uses Jupyter notebooks to demonstrate the easy integration of Grizzly in such environments and how it can be used to process and visualize data and query results interactively.

REFERENCES

[1] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking*. Springer, 61–76.

[2] Ankur Dave, Alekh Jindal, et al. 2016. GraphFrames: an integrated API for mixing graph and relational queries. In *GRADES*. ACM, 2.

[3] Joseph Vinish D'silva, Florestan D. De Moor, and Bettina Kemme. 2018. AIDA - Abstraction for advanced in database analytics. *VLDB* 11, 11 (2018), 1400–1413.

[4] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting Pandas in a Box. In *CIDR*.

[5] Steffen Kläbe and Stefan Hagedorn. 2021. When Bears get Machine Support: Applying Machine Learning Models to Scalable DataFrames with Grizzly. In *BTW*.

[6] Yinhan Liu, Myle Ott, et al. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692

[7] Andrew L. Maas, Raymond E. Daly, et al. 2011. Learning Word Vectors for Sentiment Analysis. In *ACL-HLT*. Portland, Oregon, USA, 142–150.

[8] Aisha Mohamed, Ghadeer Abuoda, et al. 2020. RDFFrames: Knowledge Graph Access for Machine Learning Tools. *Proc. VLDB Endow.* 13, 12 (2020), 2889–2892.

[9] Devin Petersohn, William W. Ma, et al. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 11 (2020), 2033–2046.

[10] Phanwadee Sinthong and Michael J. Carey. 2019. AFrame: Extending DataFrames for Large-Scale Modern Data Analysis. In *Big Data*. 359–371.

[11] Matei Zaharia, Mosharaf Chowdhury, et al. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX*.

[12] Yi Zhang, Herodotos Herodotou, and Jun Yang. 2009. RIOT: I/O efficient numerical computing without SQL. In *CIDR*.

DocDesign 2.0: Automated Database Design for Document Stores with Multi-criteria Optimization

Moditha Hewasinghage, Sergi Nadal, Alberto Abelló
Universitat Politècnica de Catalunya
Barcelona, Spain
moditha|snadal|aabello@essi.upc.edu

ABSTRACT

We present DocDesign 2.0, a novel system that supports database design for document stores. DocDesign 2.0 automatically generates a document store design driven by a query workload and a set of optimization objectives. In the presence of a massive search space, DocDesign 2.0 adopts multi-objective optimization techniques that, with high probability, guarantee to yield the optimal design based on the preferences (i.e., weights) provided by the end-user. In this paper, we demonstrate how DocDesign 2.0 improves the productivity on the task of designing a document store, as well as how the quality of the results is improved with respect to those obtained by manually generating the design.

1 INTRODUCTION

The plethora of current NoSQL systems introduces alternative data storage methods to the traditional relational database management systems (RDBMSs) [2]. Among these, document stores have gained popularity due to the semi-structured data storage model. In contrast to the RDBMS normalization, document stores favor embedding, trying to keep the data related to a single instance together instead of spreading it across different tables. This increases the complexity of database design for document stores as opposed to RDBMS, where reaching 3NF or BCNF guarantees an optimal database design in the majority of the use-cases. Database design for document stores is, in general, given low precedence, and mostly carried out in a rule-based ad-hoc manner. For instance, MongoDB, the leading document store, provides a set of design patterns[1] that provide certain guidelines on how to structure documents. However, it has been shown that the choice of design has a major impact on performance, specially in the NOSQL realm [1]. Thus, it is advantageous to have a better design by exploiting any prior knowledge on the requirements rather than a purely random one.

Let us take an example of implementing an online auction system based on the RUBiS benchmark [3] in a document store. Fig. 1 shows the 5 entities and 6 relationships composing the RUBiS framework. We can have a normalized solution, similar to that in a RDBMS, an embedded single-document solution, or the solution suggested by a purely workload-based schema recommender, such as DBSR [11], which denormalizes certain entities. To show the complexity of finding the optimal database design in a document store, let us define a running example use case consisting of two single entities from RUBiS, namely Product and Comments, and an equiprobable hypothetical workload defined as follows:

- Given a Comment ID, find its text.
- Given a Product ID, find its name.
- Given a Comment ID, find the Product name.
- Given a Product ID, find all of its Comments.

In this scenario, we have two entities and one relationship. If we assume that all attributes for an entity are kept together within a document, we are left with the decision on where the relationship must be stored in the final design. Thus, database designs can be enumerated based on the alternatives to store the relationship, which depend on three independent choices: direction, representing, and structuring as shown in Fig. 2, together with two examples. **Direction** determines which entity keeps the information about the relationship. It can be one of the two entities, or both. **Representation** affects how this relationship is stored either by keeping a reference or embedding the object. Finally, **Structuring** determines how we structure the relationship, either as a nested list or flattened. For example, if keep the references to the comments in the product, they can

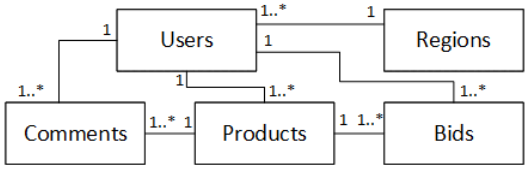


Figure 1: ER diagram of RUBiS Benchmark

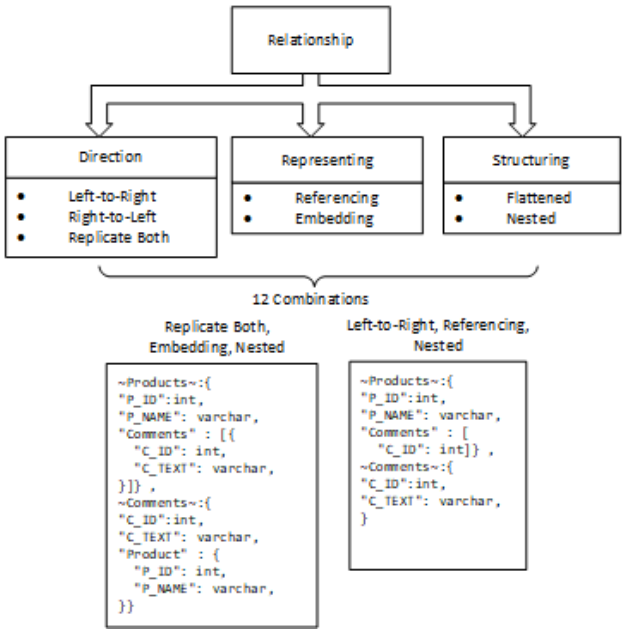


Figure 2: Relationship design choices, and two examples

[1] https://www.mongodb.com/blog/post/building-with-patterns-a-summary

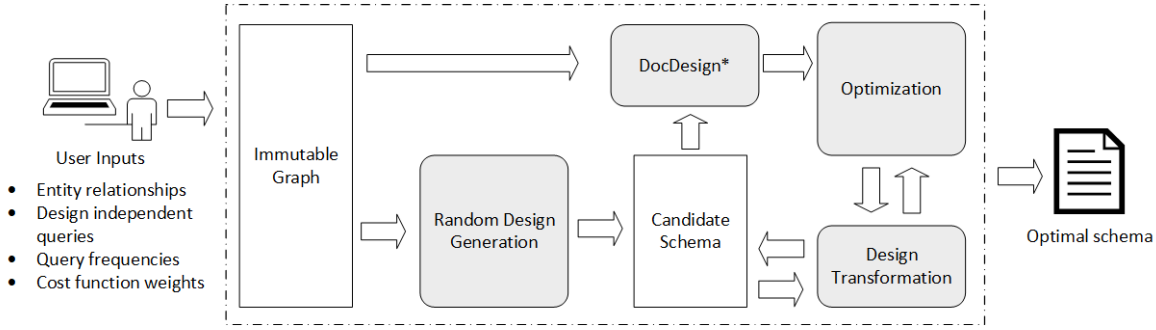


Figure 3: Overview of the DocDesign 2.0 Architecture

be stored as a list of references (*comment:..*) or in a flattened manner (*comment_1:.., comment_2:..*). Hence, we end up with 12 possible designs for our running example. Each of these could potentially be the optimal solution for an end-user depending on their preferences. For example, the design where products and comments nest their counterparts redundantly (i.e., both directions are stored by embedding the objects) will benefit query performance, as all queries can be answered with a single random access. However, this is at the expense of storage space due to redundancy. What if we only have a single reference on the product for its comments? Does the reduction of storage space justify the impact on performance? This trade-off between alternatives makes the process of finding the optimal design a complex one.

The number of relationships of the use-case (r) determines the number of candidate designs, which is exponential (12^r), as the storage option of each relationship is independent of others. Note, however, that here we did not consider allowing heterogeneous collections/lists, which is possible in the context of schemaless databases, leading to a complexity increase. For example, collections at the top level could potentially contain different kinds of documents. In our running example, user and region documents could be stored in a single heterogeneous collection mixing both. Precisely, for a design with c top-level collections, the total number of combinations will be $\sum_{i=1}^{c} \left\{ {c \atop i} \right\}$ where $\left\{ {n \atop k} \right\}$ is the Stirling number of the second kind, used to calculate the number of ways to partition n distinct elements into k non-empty subsets [5]. Overall, such exponential growth makes impossible to enumerate and evaluate all candidate designs. Hence, existing solutions, such as DBSR [11], NoSE [10] and Mortadelo [4], mainly rely on the query workload to propose a database design.

Contributions. Considering the above observations it is clear that the problem of storage design for document stores has a large search space. Moreover, each candidate solution potentially performs differently among the considered cost functions. It is, hence, obvious that exhaustively exploring the search space is prohibitively expensive. To overcome this issues, in this paper, we present DocDesign 2.0, a novel solution that addresses the complex problem of database design for document stores. DocDesign 2.0's contributions involve *automatically generating potential designs*, as well as *evaluating the performance of a design on four objectives: storage size, query performance, degree of heterogeneity, and average depth of documents*. Finally, DocDesign 2.0 *presents the end-user with the near-optimal database design specific to his/her preference of the objective* for a given use-case and query workload. Precisely, in this paper, we consider read-only query workloads. DocDesign 2.0 embeds and extends our former solution DocDesign [7], which aids on evaluating database designs based on storage size and query performance, requiring however to provide a concrete schema as input. Contrarily, DocDesign 2.0 automatically generates such designs yielding, with a high probability, the near-optimal one with respect to a set of objectives.

Outline. In the rest of the paper we introduce DocDesign 2.0's demonstrable features to resolve the motivational example and other database design for document stores scenarios. We first provide an overview of DocDesign 2.0and its core features. Lastly, we outline our on-site demonstration, involving the motivational scenario as well as other more complex real-world use cases.

2 DOCDESIGN 2.0 IN A NUTSHELL

DocDesign 2.0 adopts multi-objective optimization techniques, which have shown to be effective on obtaining near-optimal solutions out of a large search space in the presence of contradicting objectives [9]. In these scenarios, one can only aim to obtain a Pareto solution (a solution that, in the presence of multiple objectives, cannot improve one objective without worsening another).

Search algorithm. Local search algorithms consist of the systematic modification of a given state, by means of action functions, in order to derive an improved state. The intricacy of these algorithms consists of their parametrization, which is at the same time their key performance aspect. Due to the genericity of different use cases DocDesign 2.0 can tackle, we decided to choose *hill-climbing*, a non-parametrized search algorithm which can be seen as a local search, always following the path that yields higher utility values. Nevertheless, the cost functions we use are highly variable and non-monotonic, which can cause hill-climbing to provide different outputs depending on the initial state. To overcome this problem, we adopt a variant named *shotgun hill-climbing*, which consists of a hill-climbing with restarts using random initial states.

An overview of DocDesign 2.0 is shown in Fig. 3 and we present the modules and components of DocDesign 2.0 in the following subsections.

2.1 User Inputs

There are three inputs the end-user must provide, namely the equivalent to an Entity-Relationship diagram of the domain, query workload, and the weights of the cost functions.

Entity-Relationship. Refers to the use case-specific entities, their attributes, and the relationships between them. To accurately measure the different cost functions, DocDesign 2.0 requires the number of instances of each entity, the size of its

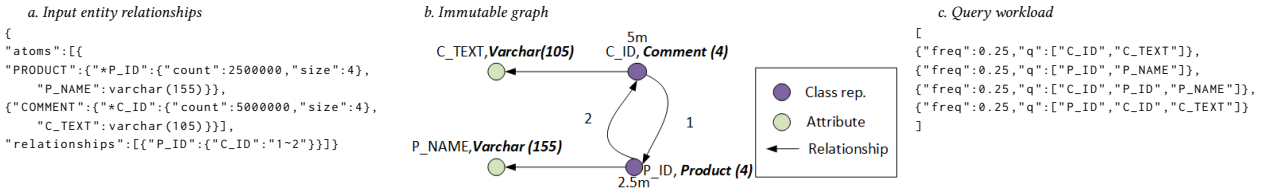


Figure 4: Input entity relationships, its internal graph representation, and query workload

attributes, and the relationship multiplicities (Fig. 4.a). This information is considered immutable, and the database design is carried out on top of it. We use a hypergraph-based canonical model internally to represent them [8]. (shown in Fig. 4.b). Furthermore, the entities are atomic, meaning that attributes related to an entity cannot be split.

Query workload. Consists of a set of queries together with their frequencies to be executed in the use case. These queries are independent of the database design and are represented as subsets of the immutable information (Fig. 4.c).

Cost function weights. Allows the end-user to include his/her preference in the database design. Currently, DocDesign 2.0 supports tuning four cost functions corresponding to the objectives: query cost, storage size, degree of heterogeneity within collections and sets, and average depth of the documents. The end-user can decide how important each of these costs are and resolve trade-offs between them. For instance, forcing higher importance to query cost and lowering the one of storage size would lead to a schema with higher redundancy and better performance.

2.2 Design Operations

Information about entities, their attributes, and the relationships are considered immutable, and the database design is built from it. Indeed, with regard to our running example, the final design must have information on all the warehouses, districts, and the relationships between them. A hypergraph-based representation enables DocDesign 2.0 to guarantee this property (we refer the reader to [7, 8] for further details). We introduce two methods to fit the shotgun hill climbing approach: generation of a random design, and evolution of a design using valid transformations.

Random design generation. The random schema generator relies on identifying subsets of entities and relationships that will be made into a collection (referred to as connected components) and the structure of the documents inside the collection in a document store database design. Based on the 12 possible designs that a relationship can be stored, we make the following decisions randomly in the schema generation process.

- **Root of the connected component** is chosen at random from the available entities. This choice determines the root document of the document store collection that this component represents. In our running example, this is either picking the warehouse or the district as the root of the collection. Let us assume we picked the warehouse in this case.
- **Choosing the next path to explore** expands the connected component and determines it's structure. Potentially multiple relationships connect an entity to others in a connected component. Thus, for a given entity of a connected component, a random subset of these relationships is picked to further expand, determining the depth and the related documents of the final design. This, together with the root of the document determines the choice of the direction in Fig. 2 except for replicating both. In the running example, we choose the relationship to the district from the warehouse (already inside the component).
- **Embedding or Reference** determines possible ways to represent the relationship between two entities of a component. If embedding is chosen, the entire document is embedded in the parent and referencing only keeps the reference of the related document on the parent. In the running example, if the embedding option is chosen, the final collection will be warehouses with embedded districts. We also make the decision of replicating both based on a given probability.

The above choices are carried out until all the entities and relationships belong to at least one of the connected components. Finally, each of the components is represented as a document store collection. These initial designs do not contain heterogeneous collections or lists, yet, since we initially ignore the choice of flattening and only use the nested option for structuring with regard to the options in Fig. 2. This decision reduces the complexity of the random generation and the number of starting schemas. However, we introduce this through design transformations to ensure that we do not lose certain designs in the process.

Design transformations. Even though it is possible to generate most of the potential designs through the random generator, it is very unlikely to reach an optimal state randomly. Moreover, we omitted the heterogeneous collections/lists and flattened ones in the random process. Thus, we introduce seven design transformation operations and use five of them to generate the neighbors of a particular design. These transformations are inspired by the rule-based design patterns proposed by MongoDB.We have validated them by recreating the MongoDB design patterns as sequences of transformations[2].

- **Union** - merges two collections/lists at the same level and creates a heterogeneous one.
- **Segregate** - separates a homogeneous collection/list out of a heterogeneous one.
- **Embed** - embeds a related document inside another.
- **Flatten** - flattens an embedded document or a list inside it's parent.
- **Group** - creates an embedded list of related documents inside another (opposite of flattening a list).

We also identify two other operations, namely, **Nest** and **Split**. Nest operation creates a nested document inside another and is unnecessary as we already cover it through the random generation. Split is similar to vertical partitioning a document. However, adhering to the atomic entity rule, we decided not to include this operation as it would also expand the search space uncontrollably.

2.3 Optimization

Candidate designs obtained through random generation or transformation need to be evaluated in order to assess their optimality.

[2]More details at https://www.essi.upc.edu/~moditha/transformations

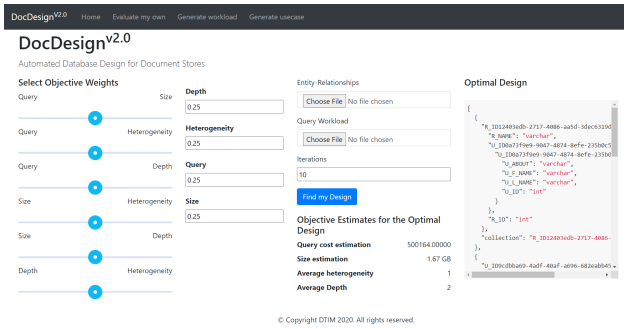


Figure 5: DocDesign 2.0 user interface

Cost functions. We introduce four cost functions to be measured and optimized in DocDesign 2.0: query cost, storage cost, degree of heterogeneity, and average depth of the documents. These are defined as follows:

- **Query cost** (CF_Q), is the sum of the relative query performance values calculated from the schema using a cost model for document stores [6].
- **Storage size** (CF_S), is the total storage size required by the collections and indexes, calculated using the canonical model.
- **Degree of heterogeneity** (CF_H), is the number of different types of documents in a collection/list. We use the average over all the collections and lists of the schema. Each heterogeneity is given a weight depending on which level the list/collection lies in the document. The higher the level, the higher the assigned weight, penalizing heterogeneities at higher levels of the document structure.
- **Depth of the documents** (CF_D), is the average depth of the documents of the design.

Utility function. Guiding the local search algorithm requires the definition of a utility function taking into account the end-user's preferences. Here, this is a function to be minimized. Hence, the end-user can assign weights to each of the cost functions according to their importance in the use-case. Then, for a given design C, we define the utility as the normalized weighted sum of each cost function $u(C) = \sum_{i=1}^{n} w_i \frac{CF_i(C) - CF_i^o}{CF_i^{max} - CF_i^o}$. The expression considers the weight w of each cost function, which is used on the transformed utility function for C. This is a normalized value that considers the *utopia* (i.e., the expected minimal) and the maximal design costs, yielding values between zero and one.

3 DEMONSTRATION OVERVIEW

DocDesign 2.0 has a web interface as shown in Fig. 5. In the on-site demonstration, we will showcase DocDesign 2.0 using the RUBiS usecase as a real-world example The manual database design process is expensive as RuBiS contains five entities and six relationships, leading to a large solution space. Moreover, we use the 11 queries with their access frequencies as the workload. First, for the ease of explanation, we will use the paper's running example (i.e., Products and Comments) and the four queries to showcase the ease of using DocDesign 2.0, initially with equal weights and then higher weight to query cost. In the first scenario with equal weights, the optimal schema is products having references to their comments. When optimizing only for the query performance, DocDesign 2.0 suggests redundantly nesting comments inside the product and product inside the comment. This approach reduces the actual runtime almost by half at the expense of double the storage space. This establishes the functionality and the efficiency of DocDesign 2.0.

Then, we will import the full RUBiS E/R to DocDesign 2.0 together with the queries and showing the ability of DocDesign 2.0 to solve more complex use-cases. The results presented by DocDesign 2.0 have a higher throughput once implemented compared to the best solution suggested by DBSR [11]. Moreover, the suggestion by DocDesign 2.0 has far less redundancy compared to the ones by DBSR. The participants are also allowed to interact with the DocDesign 2.0 demonstration with the ability to choose between different queries and objective function weights as well as generate their own. The resulting updates made to the design can be discussed by means of changes introduced (e.g.: giving more importance to query cost will result data redundancy). We also present the actual runtimes (calculated by a benchmarking suite) and storage sizes for the usecases and the designs that we demonstrate. This allows the users to validate the effectiveness of the solutions generated by DocDesign 2.0.

Since the JSON input format is specific to DocDesign 2.0, we also include a functionality to create them through an intuitive UI. Moreover, the users can suggest their own design to compare against the one suggested in terms of the four objective functions. The designs suggested by DocDesign 2.0 rely on pre-defined queries. If the queries are unknown the end users have to rely on the other three cost functions to obtain a "good enough" design. Through this hands-on experience, we are able to show the ability of DocDesign 2.0 to address the complex problem of document store database design improving the quality and productivity as opposed to a manual design process.[3]

ACKNOWLEDGMENTS

This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate Information Technologies for Business Intelligence - Doctoral College (IT4BI-DC)

REFERENCES

[1] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. 2016. Data modeling in the NoSQL world. *Computer Standards & Interfaces* (2016).
[2] Rick Cattell. 2010. Scalable SQL and NoSQL data stores. *SIGMOD Record* 39, 4 (2010).
[3] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. 2002. Performance and scalability of EJB applications. In *SIGPLAN*. ACM, 246–261.
[4] A. de la Vega, D. García-Saiz, C. Blanco, M. E. Zorrilla, and P. Sánchez. 2020. Mortadelo: Automatic generation of NoSQL stores from platform-independent data models. *Future Gen. Comp. Sys.* 105 (2020).
[5] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley.
[6] Moditha Hewasinghage, Alberto Abelló, Jovan Varga, and Esteban Zimányi. [n.d.]. A Cost Model for Random Access Queries in Document Stores (Under review).
[7] Moditha Hewasinghage, Alberto Abelló, Jovan Varga, and Esteban Zimányi. 2020. DocDesign: Cost-Based Database Design for Document Stores. In *Int. Conf. Scientific and Statistical Database Management*. SSDBM.
[8] Moditha Hewasinghage, Jovan Varga, Alberto Abelló, and Esteban Zimányi. 2018. Managing Polyglot Systems Metadata with Hypergraphs. In *Int. Conf. on Conceptual Modeling*. ER.
[9] R Timothy Marler and Jasbir S Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization* 26, 6 (2004), 369–395.
[10] Michael Joseph Mior, Kenneth Salem, Ashraf Aboulnaga, and Rui Liu. 2017. NoSE: Schema design for NoSQL applications. *IEEE Trans. Knowl. Data Eng.* 29, 10 (2017).
[11] Vincent Reniers, Dimitri Van Landuyt, Ansar Rafique, and Wouter Joosen. 2020. A Workload-Driven Document Database Schema Recommender (DBSR). In *Int. Conf. on Conceptual Modeling*. ER.

[3] Demo video available at https://vimeo.com/505248323

Visualizing and Exploring Big Datasets based on Semantic Community Detection

Maria Krommyda
School of ECE, NTUA
mariakr@dblab.ece.ntua.gr

Konstantinos Tsitseklis
School of ECE, NTUA
ktsitseklis@netmode.ntua.gr

Verena Kantere
School of ECE, NTUA
verena@dblab.ece.ntua.gr

Vasileios Karyotis
Dept. Informatics, Ionian Univ
karyotis@ionio.gr

Symeon Papavassiliou
School of ECE, NTUA
papavass@mail.ntua.gr

ABSTRACT

The extended use of the RDF model has made available many datasets from heterogeneous sources that are of interest to a wide audience. Their exploration, however, is a highly demanding task requiring extensive training and knowledge of the SPARQL language. In this demo, we present an innovative system, which supports the exploration of large RDF datasets without requiring any knowledge about the RDF or SPARQL. The system is based on a novel algorithm that detects semantically similar communities capitalizing on hyperbolic network embedding and a weighted similarity metric. The detected communities are visualized in a user-friendly way and presented to the user through a two level abstraction interface with a toolbox of exploration functionalities.

1 INTRODUCTION

Many organisations and scientists use the RDF model to share their data. There is currently high availability of linked datasets, that cover a wide range of topics and have a high degree of diversity regarding their size and characteristics [1]. Exploring and visualizing these datasets is a complex task that requires extensive knowledge about RDF and the SPARQL language. Because of their volume and update rate, they require expensive infrastructure for their processing. Therefore, even though the information is of interest to a wide audience, the datasets are, in practice, accessible only to a few data scientists.

In order to allow the exploration of linked datasets by people with no experience with the RDF and the SPARQL language, as well as no access to expensive infrastructure, we need an approach that enables: (a) accessible visualization that is scalable even for very large datasets, (b) exploration that is user-friendly and intuitive. In the following, we discuss these challenges.

Scalable visualization. RDF follows the graph structure, since the entities can be represented as nodes, and the relationships between them as edges of a graph. Systems visualizing datasets compliant with RDF, should use a graph model to represent the data, presenting each entity as a distinct node and their relationships as respective edges. Such systems should be easily accessible through commodity infrastructure and scale efficiently for very large graphs. Actually there is a requirement for efficient and scalable rendering for many devices, such as laptops, tablets and smartphones, and for a high number of simultaneous users.

User-friendly and intuitive exploration. Exploration of the dataset and extraction of relevant information should be user-friendly for users with no knowledge of the RDF model. To this end, the system should offer an easy-to-use toolbox that provides a set of exploration functionalities, such as keyword search, semantic filtering based on labels, path navigation and neighbor retrieval. Also, the system should allow interactive navigation towards the sought information. Finally, the dataset should be explorable through different abstraction levels, allowing the user to determine the degree of detail in the visualized information. Furthermore, exploration should be intuitive for users with little knowledge regarding the information that the specific dataset represents or about what can be of interest to explore.

To this end, many systems use summarization methods [6]. These can be divided in three basic categories: pattern mining, statistical and structural. Pattern mining methods employ aggregations and graph structures to identify trends in the datasets. Due to the strictness of these trends, such methods are ideal for schema identification. Statistical methods provide quantitative results over the data based on targeted queries and available semantic information. Such methods are used for the selection of the proper dataset for the user needs. The structural methods create the summaries based on the graph structure and can be further divided in quotient, which aim to identify equivalent nodes based on an equivalence relation over them, and non-quotient that use other structural measures, such as centrality, to create the summaries. Quotient summaries target indexing and querying, while non-quotient summaries are better suited for visualization and data understanding. Thus, we focus further on them.

The Grouping Nodes on Attributes and Pairwise Relationships (SNAP) [12] method is the most well-known among them. It focuses on the construction of a graph visualization that uses supernodes, nodes that contain multiple nodes of the input graph, to create summarizations based on user input and structural information such as edge values and node connections. The main drawback of this solution is the requirement for the user to select the summarization properties, in order to produce the visualized graph. Such a limitation is hindering for inexperienced users or users that want to explore datasets they are unfamiliar with.

An alternative to summarization, and a promising solution for intuitive exploration of RDF datasets, is community detection. As discussed in [7], community detection has a key role in the analysis of complex networks and the inference of useful insights regarding graph topology. However, although traditional community detection methods are very useful when applied to small networks, they cannot scale for networks of modern size as they rely on heavy computations and require a significant size of main memory. Therefore, they can process networks of up to only a few thousand nodes and edges. Hence, in order to apply community detection to RDF datasets, we need new scalable and efficient algorithms that use persistent memory and data management models to process larger graphs.

Contributions. We propose the demonstration of a robust system that implements a novel visualization technique over a novel community detection algorithm to detect communities that include semantically similar content in large RDF datasets. The system takes as input a semantically annotated dataset, divides it into semantically similar communities using a novel algorithm, stores the communities and their interconnections in a graph database and visualizes the results with respect to the graph structure. The dataset is then presented to the user through a user-friendly interface that offers two abstraction layers, allowing the user to explore both the detected communities and the subgraphs within. The system has strong advantages related to:

Semantic community detection. We have designed and implemented a novel algorithm that detects semantically similar communities [9], by first transforming the RDF dataset into a weighted graph and then employing embedding of the graph in the hyperbolic space. This is a proven "natural" space for embedding large graphs, with a semantic similarity metric aiming to detect semantically similar communities.

Community visualization. The system implements a novel two-level visualization approach: first, the nodes and edges within a community are visualized as an independent two-dimensional graph; second, the connections between communities are visualized to provide a high-level overview.

Scalability. We ensure the scalability of our technique in two ways. First, our algorithm for community detection is the first of its kind to employ a DBMS. We have selected a graph DBMS for indexing and storage of the RDF dataset to facilitate the storage of node coordinates in the hyperbolic space and the computation of the distance for all node pairs. Second, we employ a graph DBMS for storage and indexing of the inner and intra communities graphs, a design decision so that the system can support the retrieval of the information using a client-server architecture. In practice, this ensures that the system can work efficiently on any web browser and device.

Exploration functionalities. We provide an easy-to-use toolbox of functionalities that allows the users to explore the dataset using multiple filtering criteria, and navigation through panning and zooming capabilities [10].

2 SEMANTIC COMMUNITY DETECTION

We propose a new algorithm for semantic community detection. The algorithm takes as input a RDF dataset and pre-processes it in order to map it in a three dimensional (3D) space and store it as a weighted graph. In order to introduce the semantics in the detected communities we need to calculate a weight for pairs of RDF triplets that represents their semantic relation. This is based on a novel metric that encapsulates semantic and lexical similarities. Finally, the new graph is processed by the algorithm we have proposed in [9], which employs hyperbolic space concepts for semantic community detection. Algorithm 1 shows the pseudocode, and in the following we give more details for every step of the algorithm.

Step 1: RDF dataset pre-processing. Each RDF triplet of the input dataset is mapped to one node in a custom 3D space, where each one of the three dimensions correspond to *(subject, predicate, object)*. The nodes of the 3D space are connected with edges, to create a complete graph. This is the representation of the input dataset to the custom 3D space. Next, to ensure the efficiency and scalability of the algorithm for very large datasets, the information is stored and indexed in a graph database where each node has three properties.

Step 2: Weighted graph creation. In order to calculate semantically accurate weights for the graph, we follow a three-step method. First, all the words of the dataset are mapped to semantically similar groups and given a popularity score. Each node has three labels: subject, predicate and object. For each label of the node a semantic metric is calculated. Finally, the metric is used to calculate the weight between two nodes.

Initially, we aim to create groups of semantically similar words. In order to achieve this, each word is examined against all already formed groups in case it exists in one of them. In this case, the word is added to the group. In any other case, the word is examined for lexical similarity with all the words already in each group. If none is found, then the word is placed in the group that contains a semantically similar word. If no such group exists, then the word is placed in a new group.

As 'lexically similar' we consider two words that share the same lemma. For example, the words 'playing' and 'player' are lexically similar to one other, as well as to the word 'play'. As 'semantically similar' we consider two terms that have common semantic content, based on the likeness of the meaning between them, as defined in dictionaries. Two entities are semantically similar when they are associated with what is commonly refer to as 'is a' semantic relationships which are synonymy, hyponymy and hypernymy [11]. For each group i that has been formed, a score is calculated by dividing the count of words within the group with the total number of words in the dataset.

Definition 2.1. **Popularity Score.** Let $\mathbf{G} = \{\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_k\}$ be the set of semantic groups, where $\mathcal{G}_i$ for $i = 1, 2, ..., k$ corresponds to the i-th group of words and includes the words $\mathbf{W}_i = \{\mathcal{W}_{i,1}, \mathcal{W}_{i,2}, ..., \mathcal{W}_{i,m}\}$ where each $\mathcal{W}_{i,l}$ for $l = 1, 2, ..., m$ corresponds to the l-th word in the i-th group. Let $\mathbf{C}_i = \{C_{i,1}, C_{i,2}, ..., C_{i,m}\}$ where each $C_{i,l}$ corresponds to the number of times the l-th word of the i-th group is found in the dataset. Then the popularity score for the i-th group is calculated as:

$$popularityScore(G_i) = \frac{\sum_{l=1}^{m} C_{i,l}}{\sum_{i=1}^{k} \sum_{l=1}^{m} C_{i,l}} \quad (1)$$

□

Definition 2.2. **Semantic Similarity.** Each label of the node is split into m words and each word $W_{i,l}$, where $l = 1, 2, ..., m$, is replaced by the popularity score of the group it belongs to, G_i. The sum of the popularity scores divided by the number of words in the label is the semantic metric:

$$semanticMetric(label) = \frac{\sum_{n=1}^{m} popularityScore(G_n)}{m} \quad (2)$$

□

Then, for each pair of nodes the distance between them for each dimension is calculated as well as a total distance between them, the weighted average of the three distance values. The calculated distance is used as the graph weights.

Step 3: Community Detection. The community detection starts with pruning the edges of the complete weighted graph by applying the DMST method [5] to obtain a graph that is dense enough to represent the relationships between nodes and sparse enough to highlight the underlying community structure. This graph is then embedded in the hyperbolic space using Rigel Embedding [15]. In the embedding process, each node is assigned coordinates in the hyperbolic space and these are stored in the database. Based on these coordinates the computation of Hyperbolic Edge Betweenness Centrality (HEBC) follows. The HEBC

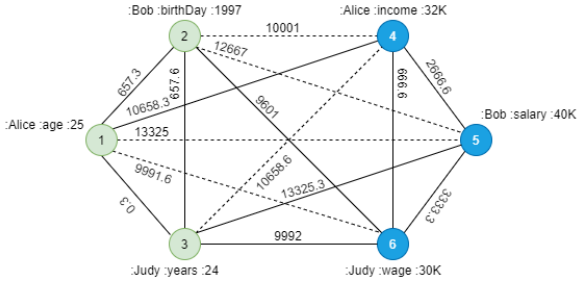


Figure 1: Semantic Community Detection

metric can be considered as the "hyperbolic" analog of the traditional Edge Betweenness Centrality (EBC) metric. The HEBC quantifies for each edge, the amount of shortest paths in the graph that pass through it. In its computations it uses the hyperbolic distance between nodes. Even though the HEBC values do not always match the nominal EBC values, the ordering of the edges is very similar. The edges of the graph are ranked in descending order and in that order edges are removed in batches, until one of the following occurs. Either the graph becomes disconnected, meaning a new community is discovered, or the maximum number of edges to remove in a single round is reached. Then, if a pre-specified number of communities is reached the algorithm terminates, otherwise the previous process is repeated for the current largest connected component of the graph. The details of the basic version of the hyperbolic-embedding can be found in our previous work[9]. In [13], we have enhanced this work for efficient and scalable processing of very large graphs.

Example. In Figure 1 we present how the proposed algorithm detects the communities on a small RDF dataset. The figure shows six nodes that relate to the ages and salaries of three people, as well as the complete graph created from the first and second step of the algorithm. The distances for each node pair are calculated as follows: the Euclidean distance is used for the numerical properties and the distance based on the semantic metric is used for the rest. This dataset is split into three semantic groups, $G_1 = (Alice, Bob, Judy)$, $G_2 = (age, years, birthDay)$, $G_3 = (salary, income, wage)$, with scores $(0.167, 0.25, 0.25)$. For example, the distance between the triplets (:Alice, :age, :25) and (:Judy, :years, :24) is computed as follows. First, the distance $d('Alice','Judy') = 0$, then $d('age','years') = 0$ and finally the distance $d(25, 24) = 1$. From that it follows that the distance between these two triplets is $\frac{0+0+1}{3} = 0.3$. In the third step, the algorithm performs the DMST, removing edges shown as dashed lines, and then we proceed to the core part of community detection, which results in two communities denoted by the two different node colors. The algorithm discovers a community that corresponds to the ages of the people and a community that contains information about their salaries. It is important to note that a different averaging technique could result in a different clustering as more importance could be given to one attribute over another. As an example, if we wanted the analysis to focus more on people's names, and thus possibly discover clusters containing information about the attributes of a person, we could use an average with larger weight on the distance between names.

Experimental results. For the Facebook network provided by SNAP library [2], our community detection algorithm obtains a modularity score of 0.59. The Girvan-Newman method resulted in a score of 0.7 but it required almost double the time necessary for our approach. As discussed in detail in [9], this result proves the benefits of our algorithm when applied to real life networks and its the ability of finding high quality clusters.

Algorithm 1: Semantic Community Detection

Input: RDF dataset, distance metric, # communities cm, # spanning trees to join k, embedding parameters $params$, maximum # edges to remove per round $batch$
Output: clusters stored as communities in a Graph DB

```
D ← distances between RDF triplets
D is stored in DB
G ← DMST(D, k)
comm_found ← 0
while comm_found < cm do
    coords ← embed(G, params)
    top_edges ← HEBC(G, coords)
    i ← 0
    while i < batch & isconnected(G) do
        G remove top_edge[i]
    if not isconnected(G) then
        comm_found ← comm_found + 1
        store newly found community in Graph DB
```

3 ARCHITECTURE & FUNCTIONALITIES

As shown in Figure 2, we have developed a server-client architecture that takes as input a RDF dataset, process it using the proposed *Semantic Community Detection Algorithm*, stores the communities to a *Graph Database* and further processes the information through two dedicated modules, the *Inner Community Visualization* and the *Intra Community Visualization*. The processed information is presented to the user through the *Visualization Interface* that it is based on a novel visualization technique [10].

Graph Database Due to the structure of the RDF model and the needs of the *Visualization Interface*, the Neo4j[14] graph database is used for the storage of the dataset. For the *Semantic Community Detection Algorithm* the custom 3D graph is stored in the graph database, but now the dataset has been restored to its initial structure and stored here. Each entity has as properties its unique identifier, the community id it belongs to, the ground truth community, if available, and its coordinates within the 2D graphical representation of the community.

Intra-Community Visualization The *Intra-Community Visualization* uses the Scalable Force Directed Placement algorithm [8] for the graphical representation of the entities and their relationships that belong in each community in the 2D space. This algorithm was selected as it allows the parameterization of many attributes of the output graph including the overlapping percentage and the size of the nodes.

Inter-Community Visualization The *Inter-Community Visualization* creates an overview of the dataset by connecting the communities in a super-graph. Two communities, i and j are super-nodes that are connected with super-edges that contain information about the real edges that connect pairs of nodes (n_i, n_j), where n_i belongs to i and n_j to j. In addition, each super-node contains information about the content of the respective community. It includes the three most prevalent nodes and edges as they are calculated based on a popularity score, calculated as follows: first, the frequency of appearances of each label in the super-node is calculated; the counter is then converted to the percentage of the label appearances in the overall dataset. As an example, if the label 'happiness' appears in the dataset 1000 times and the label 'hate' just once, then for a super-node that includes the label 'hate' once and the label 'happiness' 700 times, the score for the first would be 1 and for the second 0.7.

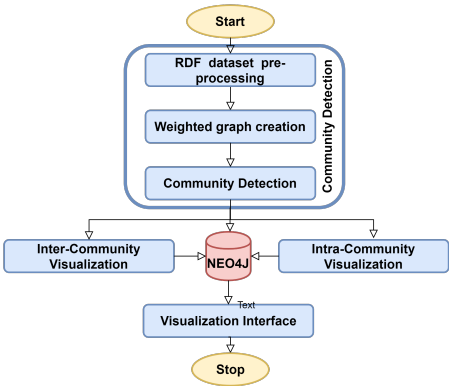


Figure 2: System Overview

This way, nodes and edges that are important to the communities but also contribute to their differentiation from the others, are included. Similarly, each super-edge has a list with the three most important edges connecting the two super-nodes.

Visualization Interface The information is presented visually through a user interface in two abstraction levels. The high level presents the interconnections between communities, as well as a brief summary regarding the context of each community. The second layer presents as a graph all the triplets of information of the community and as meta-information connections with other communities. In order to allow the user to explore the input dataset and the detected communities a series of functionalities are provided through the interface. Employing the indexing and querying capabilities of the graph database, the functionalities are provided in real-time. One key functionality is the zoom support, where the user can use the mouse wheel or the buttons at the functionality panel to visualize the graph in different levels of detail. In addition, dedicated controls allow the user to navigate between the two abstraction layers, focusing either on the overview of the communities or visualizing its information. Also, the visualization panels for the two abstraction layers are interactive and responsive to user requests. The user can browse information within them using panning and scrolling actions. The system, also, enables the user to view the information within the communities using multiple filtering and aggregation criteria either independently or at the same time. For example the user can isolate a specific edge type as well as nodes with a given node degree. To further support the exploration of the dataset, the user can locate information through keyword search. A term is matched against node and edge labels, and the result is presented as an interactive list. Last but not least, the user can select a node and focus on paths within its community, originating from it. Figure 3 shows the interface of the system, which includes the dataset selector, the functionality toolkit and the graph visualization. The graph panel depicts two of the communities based on the user selections.

4 DEMONSTRATION

We will demonstrate the prototype system that implements the proposed technique using three datasets. The audience will be able to explore the datasets through the visualization of semantic communities and compare our results with the ground truth.

Scenarios A: Semantic communities overview. The first scenario will be based on the Wikidata[3], which is a free and open knowledge base, mainly the central storage for the structured data of Wikipedia. The dataset contains a plethora of diverse information that can be used for multiple analysis based on topic, source or category. Initially, the users will be able to access multiple visualized overviews of semantic communities focusing on different attributes of the RDF entities. The overviews will focus on popular topics such as persons and places. Next, the users will delve into the fine exploration of a community graph. Given the volume and diversity of the information available in this dataset, examples of the customizable filtering and aggregation functions will allow the users to focus on specific information. The users may discover for example, based on their interests collaborations between scientists, artists or countries. Finally, the users will be encouraged to follow paths between entities aiming to identify patterns and information exchanges.

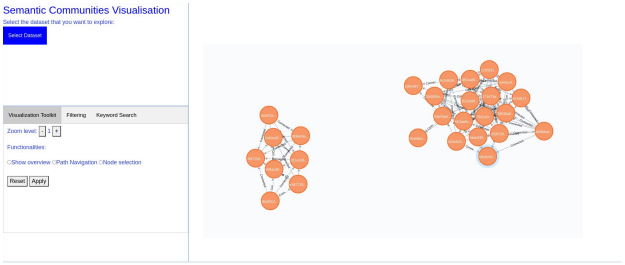


Figure 3: User interface

Scenarios B: Ground truth comparison. For the third scenario the DBLP and Amazon datasets will be used, which are offered with ground-truth by SNAP [4]. The DBLP dataset provides open bibliographic information on major computer science journals and proceedings. The RDF dataset extracted includes information about co-authors and the semantic communities are formed based on the journal or conference that a paper was published. The Amazon dataset contains a product co-purchasing network. The users will be also provided with statistics and detailed analysis of the comparison of the detected semantic communities through the proposed technique with the ground truth.

REFERENCES

[1] 2020. https://lod-cloud.net/.
[2] 2020. https://snap.stanford.edu/data/ego-Facebook.html.
[3] 2020. https://www.wikidata.org/wiki/Wikidata:Main_Page.
[4] 2020. https://snap.stanford.edu/data/#communities.
[5] Miguel Carreira-Perpiñán and Richard Zemel. 2004. Proximity Graphs for Clustering and Manifold Learning. *Adv. Neural Inf. Process. Syst.*
[6] Šejla Čebirić, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. 2019. Summarizing semantic graphs: a survey. *The VLDB Journal* (2019).
[7] Steve Harenberg, Gonzalo Bello, La Gjeltema, Stephen Ranshous, Jitendra Harlalka, Ramona Seay, Kanchana Padmanabhan, and Nagiza Samatova. 2014. Community detection in large-scale networks: a survey and empirical evaluation. *Wiley Interdisciplinary Reviews: Computational Statistics.*
[8] Tomihisa Kamada, Satoru Kawai, et al. 1989. An algorithm for drawing general undirected graphs. *Information processing letters* (1989).
[9] Vassilis Karyotis, Konstantinos Tsitseklis, Konstantinos Sotiropoulos, and Symeon Papavassiliou. 2018. Big Data Clustering via Community Detection and Hyperbolic Network Embedding in IoT Applications. In *Sensors.*
[10] Maria Krommyda, Verena Kantere, and Yannis Vassiliou. 2019. IVLG: Interactive Visualization of Large Graphs. In *IEEE ICDE.*
[11] Dekang Lin et al. 1998. An information-theoretic definition of similarity.. In *Icml*, Vol. 98. 296–304.
[12] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. 2008. Efficient aggregation for graph summarization. In *ACM SIGMOD.*
[13] Konstantinos Tsitseklis, Maria Krommyda, Vasileios Karyotis, Verena Kantere, and Symeon Papavassiliou. 2020. Scalable Community Detection for Complex Data Graphs via Hyperbolic Network Embedding and Graph Databases. *IEEE Transactions on Network Science and Engineering* (2020).
[14] Jim Webber. 2012. A programmatic introduction to neo4j. In *Conference on Systems, programming, and applications.*
[15] Xiaohan Zhao, Alessandra Sala, Haitao Zheng, and Ben Y. Zhao. 2011. Efficient shortest paths on massive social graphs. In *IEEE Collaborative Computing.*

Exploration and Analysis of Temporal Property Graphs

Christopher Rost
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
rost@informatik.uni-leipzig.de

Kevin Gomez
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
gomez@informatik.uni-leipzig.de

Philip Fritzsche
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
fritzsche@informatik.uni-leipzig.de

Andreas Thor
Leipzig University of Applied Sciences, Germany
andreas.thor@htwk-leipzig.de

Erhard Rahm
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
rahm@informatik.uni-leipzig.de

ABSTRACT

We demonstrate the *Temporal Graph Explorer*, a distributed open-source framework that enables time-dependent graph exploration and analysis on large real-world networks using a rich temporal property graph model and dynamic graph operators. In this demonstration we show how the evolution of a graph plays an essential role in many analytical questions. Besides retrieving a snapshot from a past graph state or calculating the difference between two graph snapshots, users can use our application to summarize the graph to reduce its complexity and to obtain deeper insights into its evolution. Visitors of the demo can visually experience these advanced temporal operators and their results or build and manipulate analytical programs in a declarative way without extended programming skills and run them on a distributed local or remote environment. We provide real-world temporal graph data from bicycle rentals of New York City with millions of rentals per month, also demonstrating horizontal scalability of the system.

1 INTRODUCTION

Graphs are an intuitive way to model and analyze complex relationships between entities representing real-world scenarios. Since most entities and interconnections evolve in the real-world, graphs also change over time in terms of their structure and content. For example, Figure 1 shows a toy example of bicycle rentals (represented as directed edges) between fixed stations (vertices) over time. Such temporal property graphs [1] additionally allow tracking changes in the graph over time. In the example of Figure 1 both vertices and edges store temporal information (marked by a clock symbol) as attributes (valid times) and, thus, the graph reveals that the bike with id 2115 was moved from station [1] to [2] by three consecutive rentals over time.

In this paper, we demonstrate the *Temporal Graph Explorer*, a tool for user-friendly exploration, analysis, and visualization of large temporal property graphs. The core of this application is Gradoop[1], an open-source framework for distributed graph analysis. Its *Temporal Property Graph Model (TPGM)* [8] enables modeling and analysis of graphs with bitemporal time semantics. The TPGM also comes with a set of composable temporal graph operators (including snapshot retrieval, graph evolution, and time-dependent grouping and aggregation) that can be visually configured through the user interface or programmatically

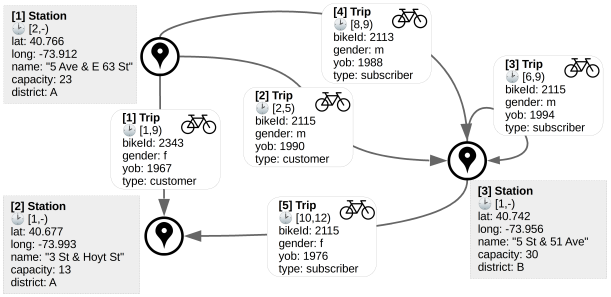


Figure 1: Example temporal property graph representing bicycle rentals between rental stations. The validity period of an edge is marked with a clock symbol and simplified with numbers instead of timestamps.

combined with the help of the declarative language GrALa [5] to build distributed analysis workflows.

The *Temporal Graph Explorer* thus enables the analysis of the evolution of graphs, i.e., to figure out *when* something happened or changed, rather than a static view representing something that happened at some time [9]. To this end, it provides temporal graph operators (including **snapshot** retrieval and graph **difference**) to compute and visualize changes, including additions and deletions, that have been occurred. This can be used in the given bicycle example to find, for a given week in the past, all rentals that have been added, removed, or remained the same.

Our *Temporal Graph Explorer* can also be employed for the analysis of large graphs by using **time-related grouping and aggregation**. This allows for a profound exploration of a graph's structure, semantics, and development over time, which is a significant part of knowledge discovery for temporal graphs. Such a graph grouping mechanism helps to find out *how* different types of vertices and edges are connected as well as *when* and *how long* they were connected. In addition, the graph can be grouped on different dimensions, e.g., by rental time or by station location, as well as on different dimension levels, e.g., per year or month for the time dimension. The grouped vertices and edges can further be aggregated in any conceivable way, from a simple count to the minimum, maximum and average duration of a specific relationship type.

2 TEMPORAL GRAPH ANALYSIS AND EXPLORATION

The *Temporal Graph Explorer* is an application to explore, analyze and visualize temporal property graphs. An intuitive web-based user interface enables the configuration of selected temporal

[1] https://github.com/dbs-leipzig/gradoop

graph operators and their application on a predefined graph dataset of the user's choice. The whole processing is then executed in a distributed environment by using Gradoop as a backend framework. The resulting graph is again temporal and sent back to the user interface for visualization. Frontend and backend application communicate through a RESTful webservice. The visual representation is adjusted to the temporal characteristics of the graph, as later explained. An architectural overview of the system is given in Figure 2.

In the following, we will dive into the single parts of the application including the used bitemporal graph data model, three selected operators, and graph visualization techniques focused on understanding the graph's evolution.

Graph model and distributed processing. The heart of our *Temporal Graph Explorer* is Gradoop [4, 5], which offers many generic operators on graphs (for pattern matching, global aggregations, etc.) that can be used within workflows for graph analysis. Workflows representing graph analytical programs can be expressed in GrALa for distributed execution. All operators are closed over the model, i.e., they take graphs as input and produce graphs. Since Gradoop is built on top of Apache Flink's Dataset-API, each operator is based on a subset of Flink's transformations (map, flatmap, join, etc.) to achieve a parallel execution and scalability to large graphs. Thus, it combines and extends features of graph analytical systems with the benefits of distributed graph processing.

To maintain the full history of a graph, including any insertion, deletion, or update of a vertex, edge, or its properties, Gradoop was extended by the recently introduced TPGM [7, 8] as a graph data model. It supports labeled, directed, multi-graphs where the vertices and edges are characterized by a unique identifier, a type label, and a set of properties, modeled as key-value pairs. In addition, each vertex and edge is extended by two time intervals I_v and I_{tx} that define a lifespan regarding to valid-[2] and transaction time dimension, which ensures a bitemporal data modeling [3]. Each close-open interval is represented by two first-order attributes t_{from} and t_{to} defining the start and end timestamp of the respective time interval. Thus, a graph of this model contains all historical and rollback information and therefore allows the exploration of its evolution and retrieving valid snapshots from the past, present or future for the valid time dimension or past and present states from the transaction time domain. To provide maximum flexibility, I_v can also be empty if no time information is available or just hold a single timestamp t_{from} if the duration of the element is not available or can be neglected, e.g., the time at which a bike station was newly built. A more detailed description and scalability evaluation of the TPGM and its operators is given in [8].

Graph snapshot and difference. To enable temporal and evolutionary queries and analysis, one data management challenge for large historical graphs is the retrieval of graph snapshots as of any time-point in the considered time domain [6]. To achieve this, we developed the *snapshot* operator that can be applied on a temporal graph instance and allows to retrieve a valid state of the graph either at a specific point in time or a subgraph that is valid during a time range. The user can configure the operator by pre-defined time-dependent predicates such as *asOf()*, *overlaps()* or *precedes()*. Furthermore, user-defined predicates, as well as helper functions to extract certain time dimensions, can be used.

[2]Valid-time is also known as application time.

An important part of the analysis of graphs is the examination of changes that have occurred between two points in time. Changes, i.e. additions, deletions, and edits, represent the evolution of a temporal graph and can be selected or aggregated in subsequent analysis steps. Therefore, we demonstrate the *difference* operator that computes a graph $\triangle G$ between two graph snapshots G_1 and G_2 by determining the union $G_1 \cup G_2$ and extending each element by a property that expresses the addition, deletion, or persistence of this element respectively. The user configures both snapshots by using time-dependent predicate functions, as described before. In addition, the desired time-dimension can be selected.

Time-specific grouping and aggregation. The maintenance of the entire history of a real-world graph entails a huge amount of graph elements. A structural grouping of the graph (also denoted as summarization) will help to reduce the overall complexity and offers deeper insights into the graph's structure, distribution, and evolution. For example, a graph with billions of vertices and edges can be first grouped by the element's label to explore the schema that reveals how the heterogeneous types are connected. In addition, temporal and content information of the grouped vertices and edges can be aggregated in many ways to get knowledge about the respective groups.

The temporal grouping operator goes further and offers a flexible mechanism to group a temporal property graph by *all* available information of the vertices and edges, especially their temporal characteristics. This is achieved by the possibility of defining a function $f(v) \mapsto k$, denoted as *key function*, that maps a vertex v (or edge) to a key k on which to group. All elements mapped to the same key are grouped together and form a new super-vertex or -edge, respectively. To simplify the specification for users, Gradoop offers predefined key functions, e.g. label() to group elements by their label, as well as helper functions, e.g., functions for extracting time-related information to summarize the graph at different temporal resolutions. Since real-world graphs are usually very heterogeneous, the application of the key functions can also be restricted to nodes or edges of a certain label (*label-specific grouping*), e.g., to group *Station* vertices by district and *Trip* edges by *gender*.

Besides the grouping itself, one main feature is to enrich the grouped elements by summarized information about the group, which can be achieved by applying pre- and user-defined aggregate functions. Not only properties but also information from the additional time dimensions can be aggregated. For example, the earliest or latest beginning of an edges validity can be calculated by using *minTime()* or *maxTime()* aggregate functions.

Analysis specification and result visualization. As already mentioned, Gradoop workflows are described in the declarative language GrALa. Besides the possibility for users to write programs directly in GrALa (see programmatic demonstration) the Temporal Graph Explorer allows the creation of simple workflows with the help of an adaptive user interface. Users can select operator(s) and the user interface allows easy parameterization by displaying suitable graphical elements, e.g. drop-down lists for selecting a snapshot predicate.

The resulting temporal graph is visualized by the Temporal Graph Explorer using Apache ECharts[3] and the JavaScript library of Cytoscape[4], an open-source software platform for visualizing

[3]https://echarts.apache.org
[4]https://js.cytoscape.org/

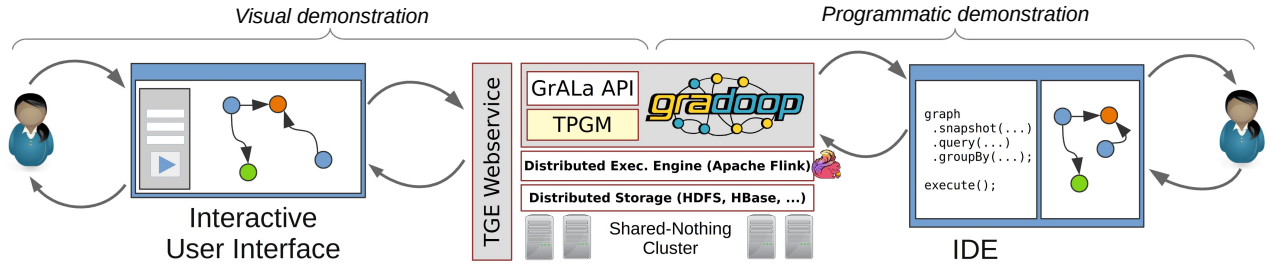


Figure 2: System architecture overview of Temporal Graph Explorer and user interaction workflow for both demonstration scenarios.

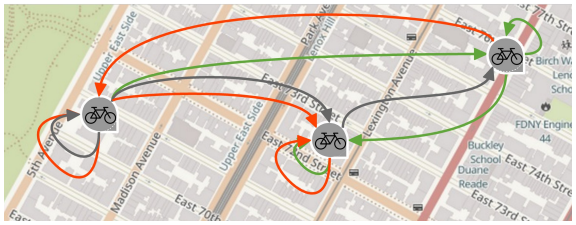


Figure 3: Example visualization of a difference graph.

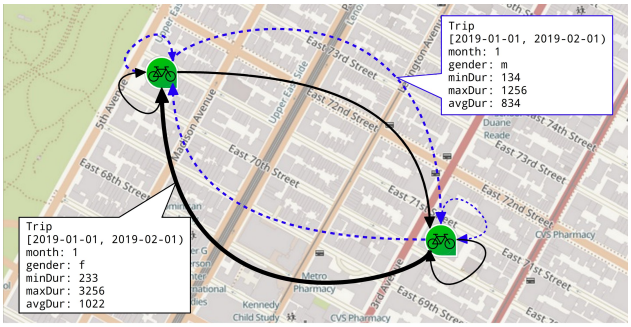


Figure 4: Visualization of two grouped stations each placed in the grid center and their aggregated trips distinguished by gender (blue dotted = male; black solid = female).

complex attributed networks. By default, the coloration of vertices and edges is based on the respective label, i.e., elements with the same label are equally colored. Further, identifiers, properties, and temporal attributes are displayed in a tooltip after selecting a vertex or an edge.

Besides this default graph visualization, we provide two additional graph representations: a difference graph view and a grouped graph view. Figure 3 shows a cutout of a difference graph's visualization. It colorizes the vertices and edges depending on the annotation with which the elements are expanded by the difference operator. A vertex or edge is colored red if the elements have been deleted in the time between both snapshots, grey if the elements were not changed at all, or green if the elements were created.

For the visualized result of the grouping operator, aggregated properties (e.g., `count`, `minDuration`, `maxTimestamp`) can be used to adjust the radius of vertices and width of edges to the corresponding property value. For example, the width of a super-edge depends on the average duration of the grouped edges. Besides, if the graph data set contains geographic coordinates as properties of vertices, these can be mapped onto an interactive map using a geo-layout (see the example in Section 3).

Further, the graph can be exported to the graph description language *DOT*, which can be easily rendered by *Graphviz* library or *Gephi*, an external visualization and exploration tool.

3 DEMONSTRATION DESCRIPTION

Our demonstration of the *Temporal Graph Explorer* is separated into two parts as shown in Figure 2. First, we demonstrate our web-based toolbox and user-interface for Gradoop. There, different analytical operators can be selected and configured in many ways, whereas the resulting graph is presented to the user by graph visualization to present the analytical value of the operators. We provide real-world graph data based on the open-source New York CitiBike[5] database which captures bicycle rentals since the year 2013. The structure of the graph is analogous to the example graph of Figure 1. The dataset consists of around 1,000 vertices (ascending over time) representing the rental stations and more than 2,000 trip edges per hour, which results in about 17 million trips per year. Further, we also integrated other temporal networks such as a heterogeneous social network synthetically generated by the LDBC data generator [2]. In the second part of the demonstration, we give visitors the opportunity to inspect and manipulate existing example temporal programs written in GrALa. All our examples can be executed on demonstration data locally, but also remotely on our research cluster.

Visual demonstration. To give more insight about results of frequently used single temporal operators, such as *snapshot*, *difference* and *temporal grouping*, we demonstrate their usage by the *Temporal Graph Explorer*. The explorer offers each visitor the possibility to parameterize the operators by using a dynamic user interface. Each operator is then executed by a Gradoop instance from where resulting temporal graphs are pushed back to the web application for visualization. The visualized graph is interactive, i.e., visitors can zoom in and out, drag vertices and edges to other positions or click on them to show their properties and temporal attributes.

To demonstrate the *snapshot* operator, a user is able to choose between supported time predicates, e.g., *asOf*, *fromTo* or *overlaps*, a time dimension (valid- or transaction time) and a respective timestamp or interval. Through the visualization, the user receives instant feedback on the changes made and thus can explore and compare various states of different times.

For the *difference* operator, a user can compare two temporal graph snapshots by exploring a difference graph. To define the

[5]https://www.citibikenyc.com/system-data

```
1  bikeGraph
2   // get all elements that overlap year 2019
3   .snapshot(
4    new Overlaps(
5     LocalDateTime.of(2019, 1, 1, 0, 0),
6     LocalDateTime.of(2019, 12, 31, 23, 59)))
7   // remove dangling edges
8   .verify()
9   // filter edges by year of birth
10  .subgraph(e -> e.getProperty("yob") > 1990)
11  // summarize the graph
12  .groupBy(
13   // group vertices by label and grid id
14   [
15    label(),
16    v -> getGridId(v)
17   ],
18   // do not aggregate vertices
19   [ ],
20   // group edges by label, month and gender property
21   [
22    label(),
23    timeStamp(VALID_TIME, FROM, ChronoField.MONTH_OF_YEAR),
24    property("gender")
25   ],
26   // calc min, max and avg duration for grouped edges
27   [
28    new MinDuration("minDur", VALID_TIME),
29    new MaxDuration("maxDur", VALID_TIME),
30    new AverageDuration("avgDur", VALID_TIME)
31   ]
32  )
```

Figure 5: Analytical application defined in GrALa.

snapshots, we are using the same implemented time predicates. For the visualization, graph elements are colorized to provide more information about their temporal evolution as described in Section 2. Using this kind of visualization a user gains insights about how and how frequently the graph evolves between two graph states.

Temporal grouping, the last operator we demonstrate, offers a large variety of possibilities to explore the temporal graph by summarizing it. The configuration options of this operator are very extensive and depend on the characteristics of the selected graph dataset and the objectives of the corresponding analysis. The *Temporal Graph Explorer* supports the user in the configuration of the operator by a flexible selection of the predefined key functions for vertices and edges, as well as aggregate functions. Appropriate arguments are offered for parameterized key functions. For example, a list of property names is offered for the function `property(<name>)`. Timestamps which appear to be useful for the selected temporal graph are also suggested for use with temporal key functions. Besides, the user can choose from pre-defined aggregate functions to additionally configure the grouping and thus to enrich the grouped elements with detailed information about the grouped element, which can be accessed in the graph visualization. The user can thus interactively add key and aggregate functions to the configuration step by step until the grouped graph and its aggregated values provide information about a specific analysis question. Figure 4 shows a cutout of a grouping result in the *Temporal Graph Explorer*. The used configuration is equal to the later-described analytical application. Since the grouped vertices have geographic properties, they can be placed on a map-view. Edges can also be colored according to a certain property, as one can see in the figure. The properties created by the aggregates are displayed after selecting a vertex or an edge.

Programmatic demonstration. For a better understanding of the API of our temporal operators, we prepared a set of example programs[6] that show basic functionality and usage. Advanced examples, like the one in Figure 5, demonstrate the composition of multiple (temporal-) operators to answer specific analytical questions.

In an example scenario, we want to answer the following question: *In 2019, how did the minimum, maximum and average trip duration change per month for male and female users born after 1990 between stations located in different areas?*

We first use the *snapshot* operator (line 3) to retrieve all information about trips, users, and stations of the whole year of 2019. Since *snapshot* can produce dangling edges, we make sure to remove them by calling the *verify* operator (line 8). We further apply a *subgraph* operator with specific predicates (line 10) to filter for users born after 1990. At the end of our pipeline, we want to summarize our graph by calling the *grouping* operator with specific grouping key functions for vertices (lines 15-16) and edges (lines 22-24). Besides the predefined *label()* function, we also show the usage of a user-defined grouping key function (line 16). It calculates a map grid using latitude and longitude properties of our vertices. We further group the edges for each month of the year, separated by the two genders of the users. During this step, we also apply multiple aggregation functions to calculate the minimum, maximum, and average duration of trips (lines 28-30). Figure 4 shows the results of this GrALa program for two randomly picked stations.

The example illustrates the level of abstraction using our operators. A user does not have to care about the underlying graph data structure, operator implementation, or distributed execution details. Gradoop offers a variety of *DataSources* and *DataSinks* to read and write different kinds of data such as *.csv* files for data exchange or *.dot* files for graph visualization.

A visitor of this demonstration has the possibility to freely manipulate the provided GrALa programs. For example, it is possible to use *asOf* instead of *overlaps* at the *snapshot* operator (line 3) to specify a different time period. Further a user is able to use *pattern matching* instead of *subgraph* (line 10) to detect important graph patterns. Also the *grouping* operator can be parameterized by different pre- or user-defined functions to summarize different aspects of the data.

ACKNOWLEDGEMENT

This work is partially funded by the German Federal Ministry of Education and Research under grant `BMBF 01IS18026B`.

REFERENCES

[1] Charu Aggarwal and Karthik Subbian. 2014. Evolutionary network analysis: A survey. *ACM CSUR* 47, 1 (2014), 1–36.
[2] Orri Erling et al. 2015. The LDBC social network benchmark: Interactive workload. In *Proc. SIGMOD*.
[3] Tom Johnston. 2014. *Bitemporal data: theory and practice.* Newnes.
[4] Martin Junghanns et al. 2016. Analyzing Extended Property Graphs with Apache Flink. In *Proc. SIGMOD NDA Workshop*.
[5] Martin Junghanns, Max Kießling, Niklas Teichmann, Kevin Gómez, André Petermann, and Erhard Rahm. 2018. Declarative and distributed graph analytics with Gradoop. *PVLDB* 11, 12 (2018), 2006–2009.
[6] Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *Proc. ICDE*. IEEE, 997–1008.
[7] Christopher Rost, Andreas Thor, Philip Fritzsche, Kevin Gómez, and Erhard Rahm. 2019. Evolution Analysis of Large Graphs with Gradoop. In *Proc. ECML PKDD*. Springer, 402–408.
[8] Christopher Rost, Andreas Thor, and Erhard Rahm. 2019. Analyzing Temporal Graphs with Gradoop. *Datenbank-Spektrum* 19, 3 (2019), 199–208.
[9] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. 2019. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering* 4, 4 (2019), 352–366.

[6] https://git.io/JvSu0

Coronis: Towards Integrated and Open COVID-19 Data

Georgios M. Santipantakis
Dept. of Digital Systems
University of Piraeus
Piraeus, Greece
gsant@unipi.gr

George A. Vouros
Dept. of Digital Systems
University of Piraeus
Piraeus, Greece
georgev@unipi.gr

Christos Doulkeridis
Dept. of Digital Systems
University of Piraeus
Piraeus, Greece
cdoulk@unipi.gr

ABSTRACT

Motivated by the global unrest related to the COVID-19 pandemic, this demo paper presents a system for acquisition of COVID-related data from different, public sources, and interlinking under a common semantic data model at a fine level of granularity. The integrated data set contains data from several European countries, which come in different schemata, formats, granularity, and data integration acts as a facilitator towards querying data from different sources, joint data analysis, and identifying correlations at varying geographical level. Moreover, our work shows how such an integrated data set can be exploited to answer complex questions for the pandemic, also in combination with other data sets via federated queries.

1 INTRODUCTION

The COVID-19 virus outbreak presents major problems worldwide, as it affects every country in the world, both economically and socially. Governments publish COVID-related data daily, and it is commonly accepted that analyzing this data may unveil hidden patterns and aid in developing a better understanding of the pandemic. However, published data comes in different schemata, formats, granularity, which prevents data analysts from applying advanced spatio-temporal analysis methods for monitoring the evolution in space-time, due to the well-known problem of big data integration from disparate sources [7]. This motivates our work for building an extensible system based on linked data principles that allows integration of data that combines reports regarding COVID-19 cases from various countries with other public data sources.

To the best of our knowledge, the public data sets related to COVID-19 that are currently available, either report the total number of cases per country or per administrative region within a specific country. Typical examples of the first category are the World Health Organization (WHO) Coronavirus Disease Dashboard [6], the WorldOmeter [1] and the Johns Hopkins University dashboard [2], built mainly for monitoring the situation at a coarse level of detail, rather than for supporting any kind of analysis, as epidemiologists do with elaborated models. On the other hand, several countries report details about the confirmed cases in their administration regions [3]. Nevertheless, it is hard to extract the time series of reported cases for each country from the unstructured text. Also, summaries of reports provided per region through online repositories by individual countries, such as [4] and [5], do not share a common schema and use different data formats (JSON, CSV or ESRI shapefiles), which hinders joint data exploration and analysis. However, data integration needs to solve several problems at a technical level, such as different languages, text encodings and region identification systems used. For example, Germany uses "landkreis" codes, whereas Austria uses "Gemeindekennziffer" (GKZ) codes. As a result, it is necessary to convert this spatial data to a level of granularity that allows correlation analysis.

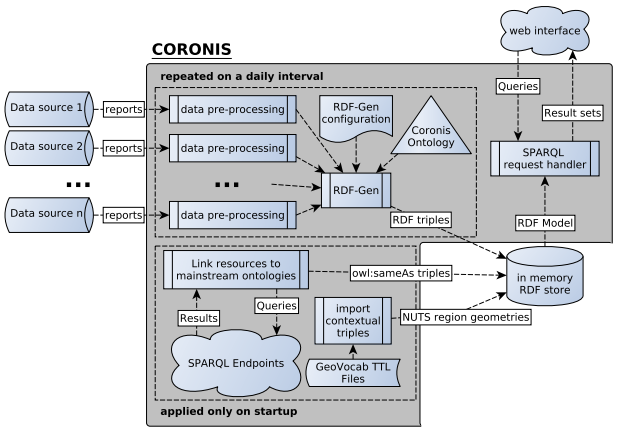


Figure 1: The overall workflow for retrieving, transforming updating and publishing data as RDF triples.

This work contributes a system, called *Coronis*[1], for gathering and linking COVID-related data from different sources under a common ontology, at a fine level of granularity and as five-star Open Data[8], enabling the correlated analysis of such data via queries that can extract useful knowledge and insights. Currently, our prototype automatically retrieves data from daily reports regarding COVID cases from 7 European countries, populating a single ontology, at a specific level of spatial and temporal granularity. Moreover, the administrative regions reported in the data set, are related to data regarding population density per region and per various age groups[2], as well as to external Open Data portals. Data exploration is enabled by a SPARQL[11] endpoint that supports federated queries. The result set is visualized using a variety of options including tables, charts and a map-based interface.

2 SYSTEM ARCHITECTURE

Figure 1 illustrates the overall system architecture as well as the workflow for data integration. Our system for COVID-19 data acquisition, integration and querying comprises the following main components:

- *Data connectors*: enabling data acquisition from different data sources.
- *Data transformation*: converts incoming data into a common semantic representation (RDF triples) according to a given ontology.

[1] In greek mythology, Coronis is a Thessalian princess and a lover of Apollo, also the mother of Asclepius, the Greek god of medicine.
[2] Data retrieved from: https://ec.europa.eu/eurostat/

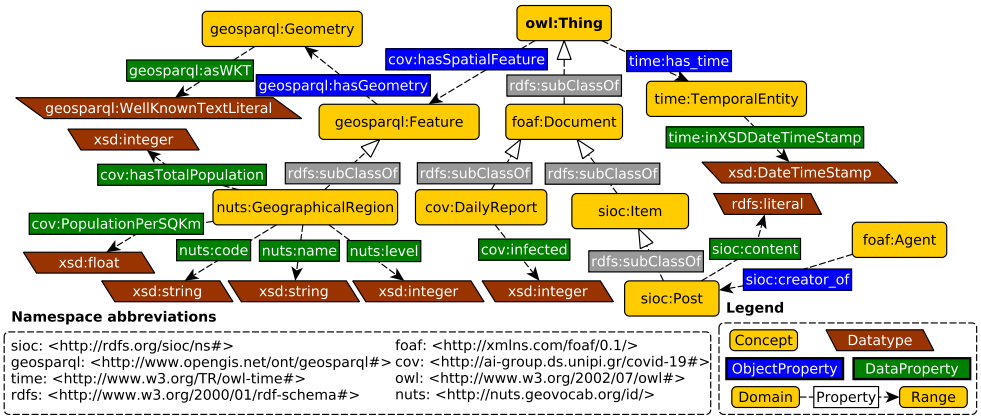


Figure 2: Core concepts (rectangles) and properties (labeled edges) of the ontology.

- *Data storage*: using an in-memory RDF store for efficient SPARQL query processing.
- *Query handler*: a SPARQL request handler that validates SPARQL queries and directs them to the RDF store.
- *User interface*: the web-based, graphical user interface that renders query results in different formats.

2.1 Data Connectors

We separate data acquisition from the remaining components, by implementing a set of data connectors, one for each data source. A data connector is responsible for establishing the connection to a remote data source, parse data, and perform data preparation tasks, such as conversion of spatial and textual encoding, build population groups, etc. The separation of data acquisition and parsing from data transformation to RDF, makes the system extensible, flexible and robust to data source modifications, or even failures of individual connectors.

The data sources accessed daily for confirmed cases per region include: Austria[3], Belgium[4], France[5], Germany[6], Greece[7], Italy[8], and Sweden[9]. Regions referenced in the data of these sources are converted to the corresponding level of *Nomenclature of Territorial Units for Statistics* (NUTS) regions. This conversion allows the correlation of regions of different countries, and also enables data integration with population data per region and age group, provided by Eurostat[10].

2.2 Data Transformation to RDF

The transformation of data into RDF triples is performed using RDF-Gen [10], our tool for efficient and flexible data transformation to RDF. RDF-Gen transforms input data using a *template* of triples that allows the use of variables or predefined functions on any of the constituent parts (subject, predicate, object) of a triple. The connector to a COVID-19 data source is initialized with contextual data related to the source, such as Administrative Regions and population (total and groups per age). The data are then automatically converted to RDF triples by RDF-Gen, and the whole process is repeated on a daily interval.

The transformation process also computes `owl:sameAs` triples between resources referring to regions in our data set and resources referring to the same regions on Open Data, such as the EU Open Data Portal, wikiData[11] and FactForge[12].

2.3 Triple Store and Query Handler

The generated RDF triples are preserved in an in-memory RDF store (Jena 3.14), which enables efficient evaluation of federated SPARQL queries over the integrated data set. In federated queries, a portion of the query is directed to a particular remote SPARQL endpoint and results returned to the federated query processor are combined with results from the rest of the query. The RDF store is initialized with static data (GeoVocab TTL files) that describe the geometries of NUTS regions and their topological relations.

The Query Handler implements SPARQL 1.1 protocol and enables our endpoint to participate in federated queries. Queries to the RDF store are supported by means of YASGUI [9], which features a user-friendly SPARQL query editor and allows rendering the result set in a wide range of formats, varying from plain CSV tables to 2D-3D maps, enriched with HTML formatted pop-ups.

2.4 The Coronis Ontology

To support the process of data integration, we build an ontology that describes the domain. Figure 2 illustrates the core concepts and properties of the ontology, where the rounded rectangles represent concepts, while edges and skewed parallelograms illustrate properties and datatypes respectively. Our ontology imports at the conceptual level:

SIOC (Semantically-Interlinked Online Communities) Core Ontology[13], OGC GeoSPARQL standard[14], OWL-Time ontology[15], and RAMON geographic ontology[16]. The integration of COVID-19 reports from different countries with EU NUTS/RAMON introduces the spatial dimension to the data, and allows the detection of topological relations between regions.

We use `cov:`, `geosparql:`, `nuts:`, as prefix abbreviations for the namespaces of our ontology, GeoSPARQL and EU NUTS RDF ontologies, respectively. The concept `cov:DailyReport`

[3]https://www.drawingdata.net/covmap/
[4]https://epistat.wiv-isp.be/
[5]https://www.data.gouv.fr/fr/datasets/donnees-hospitalieres-covid-19/
[6]https://corona.rki.de
[7]https://github.com/iMEdD-Lab/open-data/tree/master/COVID-19
[8]https://github.com/pcm-dpc/COVID-19
[9]https://visalist.io/emergency/coronavirus/sweden-country
[10]http://ec/europa.eu

[11]https://www.wikidata.org/
[12]http://factforge.net/
[13]https://www.w3.org/Submission/sioc-spec/
[14]http://www.opengis.net/ont/geosparql
[15]https://www.w3.org/TR/owl-time/
[16]https://ec.europa.eu/eurostat/ramon/ontologies/geographic.rdf

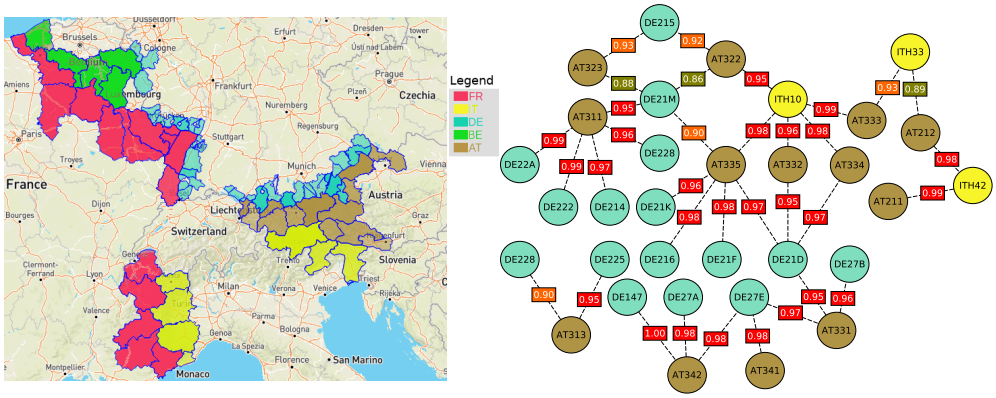


Figure 3: Adjacent regions (left) and correlations between German, Austrian and Italian regions (right).

represents the set of reports of confirmed cases. A daily report has both spatial and temporal constituents: it is associated to a resource representing a region in the EU NUTS RDF vocabulary by the property `cov:hasSpatialFeature`, and it is also related to exactly one temporal resource via the property `time:has_time`. We also define `nuts:GeographicalRegion` as a subclass of `geosparql:Feature`.

In addition, we specify a set of data properties related to the population profile of any region. Specifically, the total population, the population density, and the population per age group, with domain `nuts:GeographicalRegion`. Finally, the properties `cov:infected` and `cov:deceased` specify the number of infected and deceased cases for a region, respectively.

2.5 Queries

The Coronis ontology enables the retrieval of interlinked data with simple SPARQL queries. For example, the reported cases for a specific region (e.g., Ravensburg), sorted by date, can be obtained with the query:

```
PREFIX : <http://ai-group.ds.unipi.gr/covid-19#>
PREFIX nuts: <http://nuts.geovocab.org/id/>
PREFIX time: <http://www.w3.org/2006/time#>
SELECT ?report ?date ?population ?infected ?deceased
WHERE {
   ?report :hasSpatialFeature ?region ;
      :infected ?infected ; :deceased ?deceased ;
      time:has_time/time:inXSDDateTimeStamp ?date .
   ?region :totalPopulation ?population ;
      nuts:name "Ravensburg" .
}order by ?date
```

Figure 3 depicts an example of exploiting interlinked data from European regions, in order to investigate whether the number of infections reported in adjacent regions (of different countries) show a linear correlation. Figure 3 (left) illustrates the 67 pairs of regions returned for this query. Interestingly, the results show high correlation (0.86–1.00) between Austrian and German regions, depicted in Figure 3 (right). Lower correlation (0.25–0.46) is observed between French and Belgian regions, while negative values (-0.04 – -0.48) between French and Italian regions (probably a result of measures in Italy, when the reported number of infections dramatically increased).

To identify the adjacent regions $g1, g2$ in queries, we use the spatial predicate $touches(g1, g2)$. The Pearson R coefficient in Figure 3 (right) is computed from the results of the query:

```
PREFIX : <http://ai-group.ds.unipi.gr/covid-19#>
PREFIX nuts: <http://nuts.geovocab.org/id/>
PREFIX geosparql: <http://www.opengis.net/ont/geosparql#>
PREFIX f: <java:SPARQL_functions.>
SELECT ?r1 ?r2 ?inf1 ?inf2 ?date WHERE {
   ?r1 nuts:name ?name ;
      geosparql:hasGeometry/geosparql:asWKT ?wkt ;
      nuts:level ?l1 .
   ?r2 nuts:name ?name2 ;
      geosparql:hasGeometry/geosparql:asWKT ?wkt2 ;
      nuts:level ?l2 .
   ?c1 :hasPart ?r1 ; nuts:level "0" .
   ?c2 :hasPart ?r2 ; nuts:level "0" .
   ?rp1 :hasSpatialFeature ?r1 ; :infected ?inf1 ;
      time:has_time/time:inXSDDateTimeStamp ?date.
   ?rp2 :hasSpatialFeature ?r2 ; :infected ?inf2 ;
      time:has_time/time:inXSDDateTimeStamp ?date.
   FILTER(f:touches(?wkt,?wkt2)&&(?c1!=?c2) &&
      ((?l1="2")||(?l1="3"))&&((?l2="2")||(?l2="3")))
} ORDER BY ?date
```

3 SYSTEM DEMONSTRATION

Coronis provides a wide range of options for query building and rendering the result set, by supporting federated queries and a user-friendly web interface based on YASGUI. In this section, we provide the demonstration scenario briefly[17]. The system prototype uses a SPARQL editor to query the integrated data set.

Illustrate results in tabular format: In this option, the result set is presented as a table where each column corresponds to the projected variables, and each row corresponds to the combination of values that match the query pattern. The web interface enables sorting the results by values of specific columns or filtering the results by value. A table reporting the total number of confirmed cases for a specific date is shown in Figure 4.

Illustrate results in a grid: The result set of queries are rendered in a grid, where the cells are HTML formatted blocks dynamically constructed from the result set. For example, Figure 5 illustrates the result set of a federated query, combining number of hospitals (from wikidata) and confirmed cases (from the locally stored RDF triples) per region.

Illustrate results in a chart: In this option, a result sets that contain numerical values is presented in one of various chart types. The user can select the type of chart (and customize) by clicking on "configure". The chart can be also downloaded as an SVG file for offline use. Figure 6 depicts the number confirmed cases in Bayern per day.

[17]The queries presented in this paper and additional examples are publicly available at the endpoint's URL address: http://83.212.169.101/datasets/yasgui.html

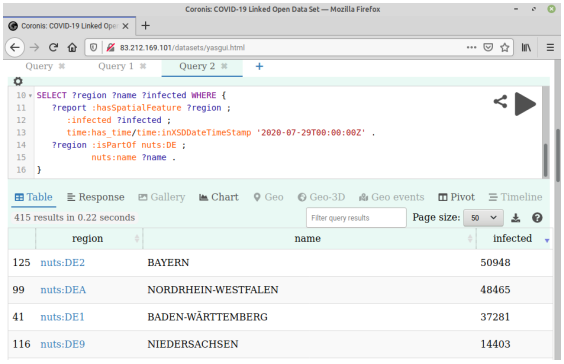


Figure 4: Result set rendered as a table.

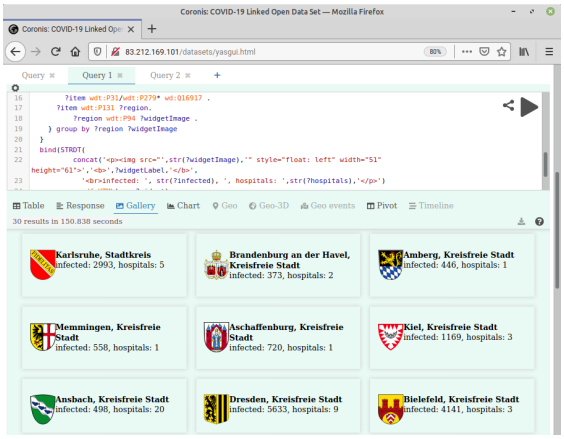


Figure 5: Result set rendered as a gallery.

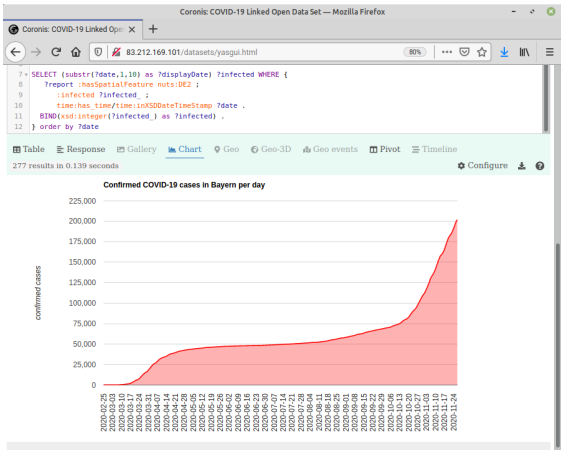


Figure 6: Example of a result set as a chart.

Illustrate results in a map: This option can be applied on results sets with spatial dimension. It renders each spatial object in the result set as a point or polygon on a 2D or a 3D map, on top of OpenStreetMap layer. The 2D map of Figure 3 and 3D map of Figure 7 are examples of this option.

4 CONCLUSIONS

This work presents Coronis, a prototype for data collection and integration of an Open Data set about COVID-19 confirmed cases

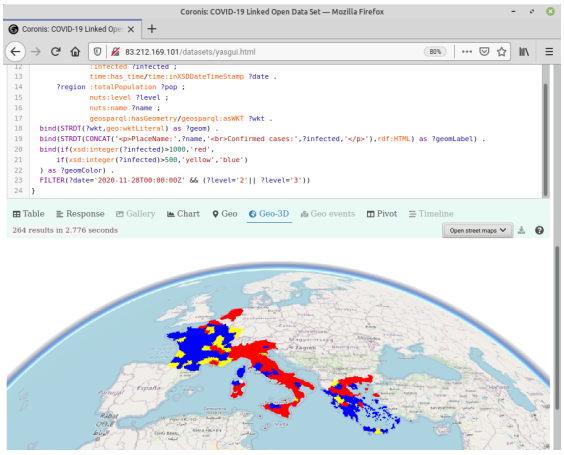


Figure 7: Result set rendered as a 3D map.

that enables cross-country, spatio-temporal analysis at different levels of granularity. Our work facilitates querying and analyzing data from different data sources, which would otherwise be a tedious and time-consuming task. The integrated data set is transformed into RDF triples to populate an ontology built on top of well-known ontologies, and resources are linked to external Open Data repositories. In turn, this enables the formulation of complex queries over interlinked COVID data with external sources, thus offering the opportunity for advanced data analysis. In our future work, we plan to expand our data set with more countries and link the data with more portals that provide information about social events, news feeds and human activities that possibly affect the spreading of the virus.

ACKNOWLEDGMENTS

The research work was supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant" (Project Number: HFRI-FM17-81).

REFERENCES

[1] 2020 (accessed July 20, 2020). *COVID-19 Coronavirus Pandemic.* https://www.worldometers.info/coronavirus/

[2] 2020 (accessed July 20, 2020). *COVID-19 Dashboard by the Center for Systems Science and Engineering at Johns Hopkins University.* https://coronavirus.jhu.edu/map.html

[3] 2020 (accessed July 20, 2020). *COVID-19 pandemic.* https://en.wikipedia.org/wiki/COVID-19_pandemic

[4] 2020 (accessed July 20, 2020). *iMedd webpage.* https://www.imedd.org/new-covid-19-i-watch-the-spread-of-the-disease-in-greece-and-around-the-world/

[5] 2020 (accessed July 20, 2020). *Presidenza del Consiglio dei Ministri - Dipartimento della Protezione Civile.* https://github.com/pcm-dpc/COVID-19

[6] 2020 (accessed July 20, 2020). *World Health Organization Coronavirus Disease Dashboard.* https://covid19.who.int/

[7] Xin Luna Dong and Divesh Srivastava. 2013. Big Data Integration. *Proc. VLDB Endow.* 6, 11 (2013), 1188–1189.

[8] Krzysztof Janowicz, Pascal Hitzler, Benjamin Adams, Dave Kolas, and Charles Vardeman. 2014. Five Stars of Linked Data Vocabulary Use. *Semant. Web* 5, 3 (July 2014), 173–176.

[9] Laurens Rietveld and Rinke Hoekstra. 2013. YASGUI: Not Just Another SPARQL Client. In *The Semantic Web: ESWC 2013 Satellite Events.* Springer Berlin Heidelberg, Berlin, Heidelberg, 78–86.

[10] Georgios M. Santipantakis, Konstantinos I. Kotis, George A. Vouros, and Christos Doulkeridis. 2018. RDF-Gen: Generating RDF from Streaming and Archival Data. In *WIMS.* ACM, 28:1–28:10.

[11] Emanuele Della Valle and Stefano Ceri. 2011. *Querying the Semantic Web: SPARQL.* Springer Berlin Heidelberg, Berlin, Heidelberg, 299–363. https://doi.org/10.1007/978-3-540-92913-0_8

Effective and Scalable Data Discovery with Nextia$_{JD}$

Javier Flores, Sergi Nadal, Oscar Romero
Universitat Politècnica de Catalunya
Barcelona, Spain
jflores|snadal|oromero@essi.upc.edu

ABSTRACT

We present Nextia$_{JD}$, a data discovery system with high predictive performance and computational efficiency. Nextia$_{JD}$ aids data scientists in the discovery of datasets that can be crossed. To that end, it proposes a ranking of candidate pairs according to their join quality, which is based on a novel similarity measure that considers both containment and cardinality proportions between candidate attributes. To do so, Nextia$_{JD}$ adopts a learning approach relying on profiles. These are succinct and informative representations of the schemata and data values of datasets that capture their underlying characteristics. Nextia$_{JD}$'s features are fully integrated into Apache Spark and benefits from it to parallelize the profiling and discovery processes. The on-site demonstration will showcase how Nextia$_{JD}$ can effectively support large-scale data discovery tasks with a large set of datasets the audience will be able to play with.

1 INTRODUCTION

Data-driven organizations are nowadays generating valuable insights by crossing their core data with external third party data, such as data from open data catalogs [6]. This has led to the creation of massive data repositories, or data lakes [8], of heterogeneous datasets without a proper structure or organization [7]. Yet, it is reported that data scientists spend up to 80% of their time in the process of discovering and integrating such datasets [9]. The lack of efficient strategies to automate such discovery process has a large impact on productivity. We exemplify this fact with the following scenario:

Example 1.1. *Emma is a data scientist employed by a marketing agency, hired to launch a campaign in the northern region of Spain. The objective is to find the best way to upsell a new product. To that end, Emma is provided with a reference dataset, such as that depicted in Table 1, containing the store locations and marketing channels that will be used to advertise the new product. She knows that the best strategy for this task is to use demographic data to define consumer segments, ultimately driving the kind of promotion and budget devoted to it. Thus, Emma plans to search for datasets in the agency's data lake, requiring to manually explore each dataset to find interesting ones to be crossed.*

1st Admin. Level	2nd Admin. Level	Store code	Channel
Aragon	*Zaragoza*	*ST123*	*Social networks*
Catalonia	*Lleida*	*ON456*	*Transit ads*
Catalonia	*Barcelona*	*ST093*	*Social networks*
Basque Country	*Araba*	*ON123*	*TV*
...	...	...	...

Table 1: Stores in Spain's northern region (D_{ref})

The previous example is a commonplace *data discovery* scenario. This is the process of automatically identifying and crossing relevant datasets to enable informed data analysis [1]. We put the focus on the task of discovering joinable attributes among datasets in a data lake. The problem is commonly tackled by measuring the similarities among pairs of attributes, aiming to provide the user with higher similarity pairs. We distinguish exact and approximate approaches, where there is a trade-off between their search accuracy and algorithmic complexity. Massive data lake environments, containing hundreds of datasets with thousands of attributes, require solutions with the ability to scale-up, and thus rule out exact methods. The state-of-the-art on approximate approaches to data discovery is those adopting *comparison by hash* techniques, such as MinHash [2] or LSH Ensemble [11]. These compare and predict similarities among pairs of attributes using techniques that, with high probability, hash similar elements to the same bucket (e.g., locality-sensitive hashing or locality-preserving hashing). This process is optimized by building index structures for a particular threshold, such that they allow to efficiently look up the predicted similarity. We next elaborate on applying hash-based techniques on Example 1.2.

Example 1.2. *Table 2 depicts a sample of Emma's agency data lake. She aims to automatically find datasets that will yield the best join with D_{ref}. To do so, as an indicator of a high quality join, she builds a threshold index to discern pairs of attributes with a containment similarity larger than 0.75. Then, Emma uses this index to find promising joinable pairs. Examples of the proposed pairs are D_{ref}.1st Admin Level = D_1.Area, D_{ref}.1st Admin Level = D_2.Region and D_{ref}.1st Admin Level = D_3.Product. Note, however, that the last pair is clearly a false positive, since many regions have a matching product name. Indeed, the values from D_{ref}.1stAdminLevel can also be found in datasets related to ship names, people names, places that are not from Spain, etc. This is a usual scenario in heterogeneous data lakes (i.e., files in different formats and covering different semantic topics) that tend to generate a significant amount of false positives pairs when only considering containment. The number of false positives generated by current approaches is overwhelming when working at scale.*

State of the art hash-based data discovery systems tend to optimistically propose too many candidate pairs at scale. Moreover, the arrival of new datasets requires to reconstruct the threshold indexes for efficient lookup. Such factors can overwhelm data scientists when dealing with large data lakes. Alternatively, another kind of approximate method to data discovery is the *comparison by profile* approach. These approaches extract summaries of datasets and their attributes to build a profile. Profiles are then compared to predict whether a given pair of attributes will join. Such succinct representations can be efficiently generated in a distributed fashion, and their comparison is much more efficient than comparing data values from a complexity point of view. Nevertheless, state of the art profile-based solutions, such as FlexMatcher [3] and Aurum [4], have a low quality prediction rate with respect to other approaches. This is mainly due to either the

(a) D_1 – Spain census data

Area	Total population	Persons under 18	Households with a computer
Aragon	3,017,804	23.2 %	84.1 %
Catalonia	973,764	20.9%	89.9%
Asturias	28,995,881	25.5%	89.2%
Galicia	6,045,680	22.1%	91.3%
...	...	...	...

(b) D_2 – Average life expectancy per region

Region	Life expectancy (Women)	Life expectancy (Men)
Catalonia	77.9	71.9
Galicia	82.6	77.5
Cantabria	78.8	73.3
Andalusia	81.4	76.8
...	...	...

(c) D_3 – One million products reviews

Product	Brand	Kind	Rating
Asturias	Sidra de Asturias	Cider	7.22
Catalonia	K. McRoberts	Book	8.83
Echo dot	Amazon	Smart speakers	8.3
Aragon	Ambar	Beer	8.22
Georgia	Fossil	Watch	5.4
...		...	...

Table 2: Three datasets proposed in a data discovery (D_1, D_2 and D_3)

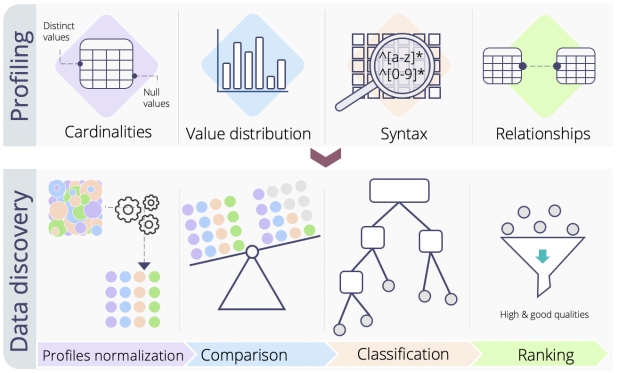


Figure 1: Profiling and data discovery stages implemented by Nextia$_{JD}$

usage of a predictive binary class (i.e., joinable or non-joinable) that generate too many false positives in practice; or to the adoption of rather basic profiles that do not accurately describe the underlying data.

In order to overcome the previous issues, we present Nextia$_{JD}$[1], a novel data discovery system with high predictive performance and computational efficiency. Nextia$_{JD}$ aims to fill the gap generated by the low predictive performance of current profile-based methods, as well as the limited precision and scalability of hash-based ones on data lakes. Nextia$_{JD}$ adopts a novel learning-based method based on data profiles. Importantly, the steps related to profiling and classification can be efficiently run in parallel. Our experiments (see [5] for more details) have shown that the predictive performance of Nextia$_{JD}$ is comparatively better than that obtained using state-of-the-art profile-based solutions, and the rate of false positives (i.e., precision) is improved w.r.t. hash-based ones. Additionally, Nextia$_{JD}$ also outperforms these systems in terms of scalability. This is achieved by integrating Nextia$_{JD}$ into the Apache Spark[2] ecosystem for distributed data processing, providing a competitive advantage with respect to the state of the art on scalability. Nextia$_{JD}$'s predictive model is based on random forest classifiers, a highly expressive model robust to outliers and noise. These models, unlike current approaches, predict a categorical quality for a candidate join based on both the containment and cardinality proportion of the involved attributes, and provide a join quality ranking that facilitates to disregard false positives.

Our demonstration will let EDBT participants impersonate Emma on her data discovery tasks. These involve exploring a data lake, generating a ranking based on the join quality of attribute pairs, as well as generating data processing pipelines from them. Similarly to other contemporary data discovery solutions (e.g., [10]), Nextia$_{JD}$ is accessible via a friendly notebook interface, nowadays the customary tool to develop and visualize data science tasks. Nevertheless, NextiaJD is able to scale-up and manage more and larger datasets. Additionally, the audience will be encouraged to try Nextia$_{JD}$ on datasets of their interest.

This will demonstrate how Nextia$_{JD}$ facilitates data discovery by reducing the time on high quality data exploration and discovery, and thus increasing the productivity of data scientists.

Outline. We next introduce Nextia$_{JD}$'s demonstrable features to resolve the motivational example and other data discovery scenarios. We first provide an overview of Nextia$_{JD}$, followed by a presentation of its core features. Lastly, we outline our on-site demonstration, involving the motivational scenario as well as other more complex real-world use cases.

2 Nextia$_{JD}$ IN A NUTSHELL

Apache Spark has emerged as the leading framework for Big Data processing due to its scalability and performance. It has been extended with modules to enable structured data processing and machine learning, namely SparkSQL and MLlib. Nextia$_{JD}$ extends Spark's source code with new operators to discover joinable datasets: `attributeProfile` and `discovery`. Figure 1, depicts a high-level overview of the stages involved in these operators.

2.1 Attribute profiling

The profiling operator implements the computation of a dataset's profile, which is composed of attribute meta-features. These represent the underlying distribution and characteristics of attributes. Hence, the method `attributeProfile` lazily computes the attribute profiles from a `DataFrame` object once and stores them for later reuse. This process can be triggered at ingestion time, or later in the discovery phase. Nextia$_{JD}$ takes full advantage of the Spark's Catalyst Optimizer to efficiently distribute the workload on very large datasets.

Kinds of profiles. Nextia$_{JD}$ collects extensive meta-features about the structure and content of String attributes in a `DataFrame`. We consider three kinds of meta-features: cardinalities, value distribution, and syntax. Cardinalities provide a broad view of an attribute via meta-features like the number of distinct values, uniqueness, or incompleteness. The value distribution builds a histogram by collecting the number of occurrences and aggregating it to compute meta-features such as the mean, standard deviation, or quantiles. Finally, syntax meta-features aim to describe the shape of data and their patterns. Nextia$_{JD}$ collects meta-features such as the length of values, numbers, or alphabetic values. Here, Nextia$_{JD}$ also exploits several regular expressions to identify specific data types such as telephones, IPs, or emails.

Overall, Nextia$_{JD}$ computes 48 meta-features that compose an attribute's profile. Figure 2, depicts the Scala code used to compute attribute profiles for D_{ref}, as well as an excerpt of its

[1]More info and resources are available at https://www.essi.upc.edu/dtim/nextiajd/
[2]https://spark.apache.org/

```
Spark.read.csv("Dref.csv").attributeProfile()
```

Attribute	Cardinality	NULLs	Entropy	% of Spaces	⋰
1st Admin. Level	17	0	12.59	5%	…
2nd Admin. Level	50	130	0.73	12%	…
Store code	8	5	8.22	1%	…
…					

Figure 2: Code to compute profiles and output's excerpt

output. Additionally, prior to the discovery process, Nextia$_{JD}$ also computes binary meta-features, which denote the characteristics of the relationship between pairs of attributes. Precisely, we measure the degree of similarity and dissimilarity between attribute names by computing the Levenshtein distance. Nextia$_{JD}$ also estimates a best-case containment scenario, assuming all unique values are covered in both attributes. The current set of meta-features used result from a principal component analysis and therefore, all of them are guaranteed to contribute with relevant information to make the decision. Indeed, Nextia$_{JD}$ computes richer profiles compared to other profile-based approaches.

2.2 Data discovery

The data discovery operator exposes the functionality to discover joinable attributes by using the profiles. We distinguish two scenarios: discovery-by-attribute and discovery-by-dataset. The former focuses on the discovery from a reference attribute, while the latter exhaustively searches all attributes in a reference dataset. Both settings require a reference dataset and a list of candidate datasets. Additionally, discovery-by-attribute requires a reference attribute name. This operator encapsulates and hides from the analyst the complexity required to implement the different stages in the discovery pipeline: profile normalization, comparison, classification, and ranking. Thus, Nextia$_{JD}$ does not require to parameterize or process the input data.

Normalization. Meta-features in a profile are represented in different magnitudes, therefore normalization plays an important role to guarantee a meaningful comparison between profiles. Nextia$_{JD}$ adopts the Z-score normalization method for all meta-features in a profile. To do so, a UDF function computes the mean and standard deviation for a given meta-feature using the respective SparkSQL aggregation functions.

Comparison. Comparing profiles requires computing distances among meta-features corresponding to a pair of attributes. Once pairs are created, we merge the profiles subtracting the normalized meta-features from the reference attribute and the to-be-compared attribute by using Spark SQL.

Classification. Nextia$_{JD}$ adopts a learning approach that allows us to classify pairs of attributes producing high quality joins. Precisely, Nextia$_{JD}$ aims to predict the join quality, which is an asymmetric rule-based measure combining both containment similarity and cardinality proportion. In [5] we introduced the concept and role of the cardinality proportion, which complements the containment metric to remove a substantial amount of the false positives generated by it. In short, the cardinality proportion contextualizes containment, and as such, different cardinalities tend to identify different semantics or granularity levels for heterogeneous data lakes. Such metric yields a quality class from a totally-ordered set $S = \{\texttt{None}, \texttt{Poor}, \texttt{Moderate}, \texttt{Good}, \texttt{High}\}$. Hence, the definition of join quality is as follows:

Definition 2.1. Let A, B be sets of values, respectively the *reference* and *candidate* attributes. The join quality among A and B is defined by the expression

$$Quality(A,B) = \begin{cases} (4)\ \texttt{High}, & C(A,B) \geq C_H \wedge \frac{|A|}{|B|} \geq K_H \\ (3)\ \texttt{Good}, & C(A,B) \geq C_G \wedge \frac{|A|}{|B|} \geq K_G \\ (2)\ \texttt{Moderate}, & C(A,B) \geq C_M \wedge \frac{|A|}{|B|} \geq K_M \\ (1)\ \texttt{Poor}, & C(A,B) \geq C_P \\ (0)\ \texttt{None}, & \text{otherwise} \end{cases}$$

Nextia$_{JD}$ embeds a set of general purpose models, one per quality label in the previous definition, trained with Random Forest classifiers from Spark MLlib. These models were trained by transforming a multi-class classification problem into a binary one per class. Each of these models takes as input the normalized unary and binary meta-features of the pair of attributes for which we aim to predict their join quality. These models were trained following good practices in model learning and the ground truth (including labeling), data preparation, and validation processes are thoroughly presented in a reproducible manner at: https://www.essi.upc.edu/dtim/nextiajd/. As part of this process, we empirically determined the values $C_H = 3/4 = 0.75, C_G = 2/4 = 0.5, C_M = 1/4 = 0.25, C_P = 0.1$ for containment, and $K_H = 1/4 = 0.25, K_G = 1/8 = 0.125, K_M = 1/12 = 0.083$ for cardinality proportion on our training dataset composed of 138 real datasets. The models validation was conducted with 139 real datasets from different topics and file sizes and yielding a high predictive performance [5]. As a result, for each candidate join pair the discovery operator associates a join quality label (i.e., from `None` to `High`) and five probability scores, one per model.

A key distinguishing factor of Nextia$_{JD}$ with regard to other profile-based approaches (e.g., FlexMatcher [3]), is that it relies on general purpose models that can be used for any data discovery process with heterogeneous datasets.

```
NextiaJD.discovery(Dref, Seq[D1,D2,D3],
"1st␣Admin.␣Level")
```

Dataset	Attribute	Quality	Probability
D_2	Region	High	0.95
D_1	Area	High	0.93

Figure 3: Code to trigger a discovery process, and the two first elements of the ranking it generates

Ranking. Finally, an evaluation is performed in the probabilities of a candidate join pair to assign a single probability. In short, the highest probability wins, except for cases where several probabilities are close to each other. In those cases, we follow a rule-based strategy to avoid misclassifications due to the fact that the probability for the `None` class is the only one predicting no join. We identified two main cases generating the most misclassifications: (i) when the no join probability (i.e., the probability for `None`) is above 50% and (ii) when the join-related labels (i.e., from `Poor` to `High`) are all below 50% and the `None` probability is close to them (measured by an empirical threshold). In these cases, the final decision is modified to the second highest probability (which in practice, given these rules, mostly means to that of `None`). Then, Nextia$_{JD}$ generates a partial order by considering, first,

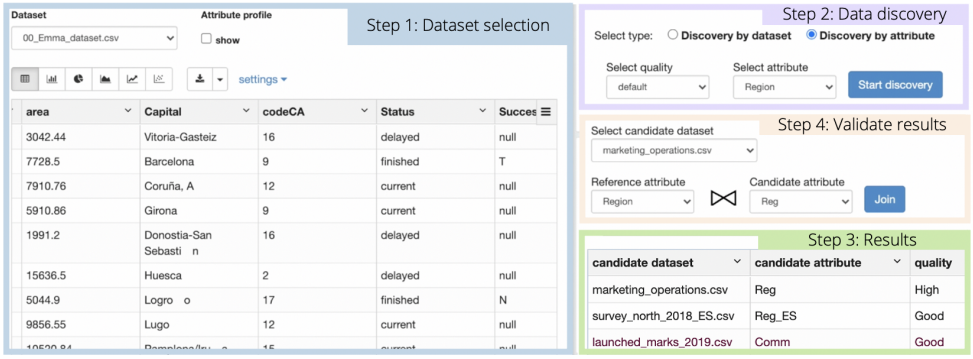


Figure 4: Nextia$_{JD}$ GUI in a Zeppelin notebook

the predicted quality label (and the totally-ordered set related to them). For pairs yielding the same quality label, we rank them considering the probability yielded by the model (i.e., that of the model of the quality labeled finally assigned). An example of the ranking produced for Emma's use case is presented in Fig. 3. By default, Nextia$_{JD}$ only shows High and Good qualities. However, other qualities can be requested on-demand.

3 DEMONSTRATION OVERVIEW

Computational notebooks have become the de-facto tool for data science projects. In our experience in several Data Science projects, data scientists like to explore datasets with their usual analytical tools. Using a third party tool to perform data discovery is disruptive for their day-by-day tasks, and they tend to avoid it. For this reason, we created Nextia$_{JD}$, which fills a current gap to bring data discovery closer to data scientists. We argue that embedding data discovery into notebooks and taking advantage of their interactive capabilities will improve data scientists' productivity. Therefore, for our demonstration, we will use a Zeppelin notebook to present the main functionalities: profiling and data discovery, and how they can be used in the day-by-day of data scientists. We have also created an informative companion website[3] where the notebooks[4], source code and experiments are publicly available. It is worthy to say that Nextia$_{JD}$ is not tied to this demonstration platform and can be integrated into any technology that supports Spark in Java or Scala clients.

We encourage attendees to impersonate Emma and follow the workflow she would have to execute using Nextia$_{JD}$. Note that we assume for this demonstration that datasets were profiled when ingested into a repository to be ready for use in further discoveries. However, new datasets can be processed and profiled on demand if required during the demo. Figure 4 shows Nextia$_{JD}$'s GUI in a Zeppelin notebook:

(1) **Dataset selection.** First, attendees can select and preview datasets available in our heterogeneous data lake containing Emma's dataset and datasets from different topics such as movies, territories, finance, etc. Through this step, attendees can also preview the profiling computed to have a better perspective of what kind of meta-features Nextia$_{JD}$ collects.
(2) **Data Discovery.** Once a dataset is selected, we proceed to the data discovery task. Users can perform two types of setting: discovery-by-dataset or discovery-by-attribute. Through this step, it is possible to select the desired quality. Nextia$_{JD}$ will execute the data discovery operator and will handle all steps: normalization, comparison, classification, and ranking.
(3) **Explore results.** After data discovery, results are visualized in a table where we show the attributes pairs found, the source of the datasets, the quality predicted, and the probability.
(4) **Validate results.** Once the attendees find an interesting pair proposed by Nextia$_{JD}$, they can validate the result by executing the join operation. This operation will update the dataset preview with the result of the join. Additionally, the similarity and cardinality proportion obtained by the join operation is also computed and shown.

Last, but not least, we are aware that some data scientists are advanced users. In these cases, they do not need to use Nextia$_{JD}$'s GUI but directly use the new it offers as an extension of Spark. These are compiled and ready in the Spark fork available from our website. This is shown in the live demo on our website. Overall, this demonstration will offer a comprehensive dive into Nextia$_{JD}$.

[3]https://www.essi.upc.edu/dtim/nextiajd/
[4]Nextia$_{JD}$ online notebooks have deactivated parallelism to keep them 24/7 in a budget machine. Find instructions to install Spark and Nextia$_{JD}$ in https://www.essi.upc.edu/dtim/nextiajd/#resources

ACKNOWLEDGMENTS

This work is partly supported by Barcelona's City Council under grant agreement 20S08704. Javier Flores is supported by contract 2020-DI-027 of the Industrial Doctorate Program of the Government of Catalonia and Consejo Nacional de Ciencia y Tecnología (CONACYT, Mexico).

REFERENCES

[1] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *ICDE*. IEEE, 709–720.
[2] Andrei Z. Broder. 1997. On the resemblance and containment of documents. In *SEQUENCES*. IEEE, 21–29.
[3] Chen Chen, Behzad Golshan, Alon Y. Halevy, Wang-Chiew Tan, and AnHai Doan. 2018. BigGorilla: An Open-Source Ecosystem for Data Preparation and Integration. *IEEE Data Eng. Bull.* 41, 2 (2018), 10–22.
[4] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. In *ICDE*. IEEE Computer Society, 1001–1012.
[5] Javier Flores, Sergi Nadal, and Oscar Romero. 2021. Scalable Data Discovery Using Profiles. In *EDBT*. To be published as short paper.
[6] Renée J. Miller, Fatemeh Nargesian, Erkang Zhu, Christina Christodoulakis, Ken Q. Pu, and Periklis Andritsos. 2018. Making Open Data Transparent: Data Discovery on Open Data. *IEEE Data Eng. Bull.* 41, 2 (2018), 59–70.
[7] Fatemeh Nargesian, Ken Q. Pu, Erkang Zhu, Bahar Ghadiri Bashardoost, and Renée J. Miller. 2020. Organizing Data Lakes for Navigation. In *SIGMOD Conference*. ACM, 1939–1950.
[8] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *Proc. VLDB Endow.* 12, 12 (2019), 1986–1989.
[9] Michael Stonebraker and Ihab F. Ilyas. 2018. Data Integration: The Current Status and the Way Forward. *IEEE Data Eng. Bull.* 41, 2 (2018), 3–9.
[10] Yi Zhang and Zachary G. Ives. 2020. Finding Related Tables in Data Lakes for Interactive Data Science. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 1951–1966.
[11] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *Proc. VLDB Endow.* 9, 12 (2016), 1185–1196.

A Tool for JSON Schema Witness Generation

Lyes Attouche
Université Paris-Dauphine, PSL Research University
lyes.attouche@dauphine.fr

Mohamed-Amine Baazizi
Sorbonne Université, LIP6 UMR 7606
baazizi@ia.lip6.fr

Dario Colazzo
Université Paris-Dauphine, PSL Research University
dario.colazzo@dauphine.fr

Francesco Falleni
Dipartimento di Informatica, Università di Pisa
fallenifrancesco98@gmail.com

Giorgio Ghelli
Dipartimento di Informatica, Università di Pisa
ghelli@di.unipi.it

Cristiano Landi
Dipartimento di Informatica, Università di Pisa
c.landi7@studenti.unipi.it

Carlo Sartiani
DIMIE, Università della Basilicata
carlo.sartiani@unibas.it

Stefanie Scherzinger
Universität Passau
stefanie.scherzinger@uni-passau.de

ABSTRACT

JSON Schema is an evolving standard for the description of JSON documents. It is an extremely powerful language endowed with boolean operators and recursive definitions. Hence, classical problems like schema *consistency* and *equivalence* may be challenging without well-principled tools. Based on our recent effort for laying down an algebraic formal semantics of JSON Schema, we demonstrate an approach for generating valid *witnesses* of a user-defined schema. Our goal is not only to allow programmers to design schemas that meet their intentions, but also to guide them in their journey to understanding the semantics of existing schemas, in an interactive fashion. We thus aim to contribute to the adoption of the JSON Schema language by facilitating its use.

1 INTRODUCTION

In recent years, JSON has become the *de facto* standard data interchange format, and is now widely used for exchanging data between web applications and remote servers, for exporting and importing data, as well as inside complex ML pipelines for combining different stages, as in Google TFX [11].

Despite its great popularity, there is no consensus about a *standard* schema language for JSON yet. Indeed, in many cases, JSON datasets come without a schema, and the end user or application has the duty to infer or guess a new schema, if required. In many other cases, however, several and vastly different schema languages are used for describing the structure of JSON data, ranging from Apache Avro [3], to the MongoDB internal schema language [6], and to JSON Schema [12].

Differently from what happened with XML, whose standard schema languages (DTDs and XML Schema) reached quickly a wide diffusion, JSON Schema is not being adopted at the same pace. Many reasons are slowing down its adoption, but, according to our observations, a major obstacle is the fact that, while extremely powerful, JSON Schema is – frankly – hard to use. Indeed, a schema is a logical combination of implicative assertions, and some of them may produce side effects on previous ones.

As a consequence, leaving the realm of plain vanilla schemas may expose the programmer to many risks, such as the definition of a schema with unintended semantics, or one that is even empty.

Example 1.1. Consider the following schema.

```
{
    "type": "object",
    "properties": {
        "x": { "type": "integer" }
    },
    "required": [ "x" ]
}
```

This schema declares that all instances are JSON objects, and that each object has a mandatory *member* whose name is x and whose type is *integer*. This schema, however, does not impose further constraints on object values; therefore, an object may also have supplementary and unconstrained members.

Example 1.2. Consider now the following schema, differing only in the next-to-last line.

```
{
    "type": "object",
    "properties": {
        "x": { "type": "integer" }
    },
    "not": { "required": [ "x" ] }
}
```

One may assume that this specifies that x is "not required", hence is optional. However, given the semantics of JSON Schema, negating a required member does not make it optional: indeed, the final effect is to actually forbid the presence of the member, hence excluding any JSON object having a member whose name is x (this example is inspired by a discussion on Stack Overflow [1], where the confusing effect of this schema is testified).

Given the complex and non-trivial interplay between schema assertions, designing a rich yet sound schema is challenging, especially when other powerful mechanisms of JSON Schema are involved, such as negation, mutual exclusion, recursion, union and conjunction, as well as array constraints controlling array length and content, possibly requiring uniqueness of elements.

Motivating Witness Generation. The state-of-the-art approach for exploring JSON Schema semantics is ultimately a manual trial and error: using a JSON Schema validator, a schema designer can test whether a JSON document is valid w.r.t. the schema. That is, the designer must come up with suitable *witness* documents.

Yet, in this demo, we present a tool capable of automatic witness generation. For instance, for the schema from Example 1.1, our tool generates the witness `{"x": 0}`, as any valid instance

must be an object, where member x is mandatory and integer-typed. For Example 1.2, our tool generates the witness `{}`, since the empty object is valid. For the schema designer, this valuable feedback may well increase the overall productivity.

Moreover, upon the push of a button, the designer can generate further witnesses. In the example just discussed, the designer would be provided with `{"0": null}`. Thus, by interactive iteration, the designer ensures that he or she "gets it right".

Moreover, our tool allows the comparison of two schemas: for a witness that is valid w.r.t. the schema from Example 1.2, but not w.r.t. the schema from Example 1.1, the tool returns the JSON document `{}`. For the other way round, a witness is `{"x": 0}`. Again, the designer can request further witnesses, as needed.

Contributions. The goal of this demonstration is to showcase a tool allowing the schema designer to investigate the formal properties of a schema, and even to compare schemas. Our tool is based on our earlier contributions on algebraic manipulations of JSON Schema [7]. With our tool, the designer can:

- obtain an algebraic representation of the input schema;
- generate a witness for the schema, to verify whether the schema is empty or not, and to gain insights into the actual semantics of a given schema;
- exploit witness generation for checking whether a schema S_1 is a subtype of a schema S_2, and hence, whether it represents a conservative and not disruptive evolution.

This combination of features is currently not supported by any existing commercial or academic tool: while tools for JSON Schema containment checking are available (e.g., [9]), they merely produce boolean answers. Ours is the first tool capable of generating actual witnesses in containment checking.

2 JSON AND JSON SCHEMA

In the following, we introduce the JSON data model and provide some intuition for JSON Schema. We refer to [7, 10] for details.

2.1 JSON data model

The grammar below captures the syntax of JSON values, which are either basic values, objects, or arrays. Basic values B include the null value, booleans, numbers n, and strings s. Objects O represent sets of members, each member being a name-value pair, and arrays A.

$J ::=$	$B \mid O \mid A$		**JSON expressions**
$B ::=$	null $\mid$ true $\mid$ false $\mid n \mid s$		**Basic values**
	$n \in \mathsf{Num}, s \in \mathsf{Str}$		
$O ::=$	$\{l_1 : J_1, \ldots, l_n : J_n\}$		**Objects**
	$n \geq 0, \quad i \neq j \Rightarrow l_i \neq l_j$		
$A ::=$	$[J_1, \ldots, J_n]$	$n \geq 0$	**Arrays**

2.2 JSON Schema

JSON Schema is a language for defining the structure of JSON documents, maintained by the Internet Engineering Task Force [4].

JSON Schema uses the JSON syntax. Each construct is defined using a JSON object with a set of fields describing assertions relevant for the values being described. Some assertions can be applied to any JSON value type (e.g., *type*), while others are more specific (e.g., *multipleOf* applies to numeric values only). The syntax and semantics of JSON Schema have been formalized in [10] following the specification of Draft-04. We limit ourself to an informal discussion about the possible constraints associated to each type:

- when defining a *string*, it is possible to restrict its length by specifying *minLength* and *maxLength* constraints, and to define the *pattern* that the string should match;
- when defining a *number*, it is possible to define its range of values (by any combination of *minimum* / *exclusiveMinimum* and *maximum* / *exclusiveMaximum*), and to define whether it should be *multipleOf* a given number;
- when defining an *object*, it is possible to define its *properties*, the type of its *additionalProperties* and the type of the properties matching a given pattern (i.e., *patternProperties*). It is also possible to restrict the minimum and maximum number of properties using *minProperties* and *maxProperties*, and to indicate which properties are *required*;
- when defining an *array*, it is possible to define the type of its *items* and the type of the *additionalItems* which were not already defined by *items*, and to restrict the minimum and maximum size of the array; moreover, it is also possible to enforce uniqueness of the items using *uniqueItems*.

JSON Schema allows the combination of assertions using standard boolean connectives: *not* for negation, *allOf* for conjunction, *anyOf* for disjunction, and *oneOf* for exclusive disjunction. A finite set of accepted values can be indicated through the *enum* constraint. Please note that hereafter, as well as in our formal development [7], we will use in some examples the usual notation for boolean operators (e.g., $\vee/\neg/\wedge$ for disjunction/negation/conjunction) and the symbols S/S_i to indicate schema fragments.

3 WITNESS GENERATION

Witness generation is challenging for JSON Schema, in particular due to the non-algebraic nature of JSON Schema (the meaning of certain assertions depends on the surrounding context), and because of the presence of negation and conjunctive schemas. We elaborate on these facts, and describe some main aspects of the formal systems and algorithms we devised. We will provide the full details in a future publication.

In a nutshell, our approach proceeds as follows. Assume that you have an algorithm to generate a witness for any schema assertion S of size up to n. In order to generate a witness for a schema of size $n + 1$ describing objects having a field of label l and value of type S, one will generate a witness w for S and use it to build an object with a field label l whose value is w.

For disjunction $S_1 \vee S_2$, we recursively generate witnesses of S_1 and of S_2. Yet negation and conjunction are problematic, as there is no way to generate a witness for $\neg S$ starting from a witness for S, and, given a witness for S_1, if this is not a witness for $S_1 \wedge S_2$, we may need to try infinitely many others before finding one that satisfies S_2. As we will see, since conjunction is used for object and array schemas, dealing with conjunction is particularly important for generating these two kinds of values.

Dealing with these challenges requires schema manipulations that can be rather complex. In order to devise the necessary schema transformation rules, as well as to study their properties and optimization techniques, we designed an algebra which is at the same time minimal and fully compliant to JSON Schema.

Details can be found in [7], but just to have a glimpse, consider the following JSON Schema fragment describing properties of label-value members of object values. In this fragment, we have a conjunction of assertions satisfied by a JSON value J if the following holds: if J is an object then i) if a k_i label is present,

then its associated value meets S_i, ii) if a label k' is present, satisfying a pattern (regular expression) r_i, then its associated value satisfies PS_i, iii) for all other labels in J not satisfying the previous conditions, the associated value satisfies S, iv) a member with label k_1 is required.

$$\begin{array}{l}\texttt{"properties"} : \{\texttt{"}k_1\texttt{"} : S_1, \ldots, \texttt{"}k_n\texttt{"} : S_n\},\\ \texttt{"patternProperties"} : \{\texttt{"}r_1\texttt{"} : PS_1, \ldots, \texttt{"}r_m\texttt{"} : PS_m\},\\ \texttt{"required"} : [\texttt{"}k_1\texttt{"}],\\ \texttt{"additionalProperties"} : S\end{array}$$

In this example, we can see the non-algebraic nature of JSON Schema: the semantics of one assertion (`additionalProperties`) depends on a co-occurring assertion (`properties`).

Our algebra encapsulates possibly interacting assertions into one, as shown below.

$$\begin{array}{l}\mathsf{props}(\underline{k_1} : \langle S_1\rangle, \ldots, \underline{k_n} : \langle S_n\rangle, r_1 : \langle PS_1\rangle, \ldots, r_m : \langle PS_m\rangle; \langle S\rangle)\\ \qquad \wedge\ \mathsf{req}(k_1)\end{array}$$

In the above algebra expression, we use $\underline{k}$ to indicate the JSON pattern "^$k$$" that only matches k, and $\langle S'\rangle$ to indicate our algebra expression corresponding to a schema S'.

By relying on this algebra, in order to deal with schemas $\neg S$ for enabling witness generation, we follow a traditional approach: we push negation inside S by playing with standard boolean laws, in order to obtain an equivalent, not-free schema that we use for witness generation. Unfortunately, JSON Schema does not enjoy negation closure: there are JSON schemas for which not-elimination is not possible. So we have extended our algebra with several new basic operators ensuring negation closure (that can be produced by not-elimination), and for which witness generation is possible in an inductive fashion, after further rewritings that we are going to exemplify. One of such operators is

$$\mathsf{pattReq}(r_1 : S_1, \ldots, r_n : S_n)$$

In order for a value to be an instance of the schema above, if the instance is an object, then, for each $i \in \{1..n\}$, it must possess a member whose name matches r_i and whose value satisfies S_i. (It is worth observing, that it is strictly more expressive than `required` since it allows one to require a name that belongs to an infinite set $L(r_i)$, and it associates a schema S_i to each required pattern r_i.)

A second challenging aspect is related to conjunction. In order to deal with conjunctive schemas we rely on standard rewritings, enabling the transformation into equivalent schemas in Disjoint Normal Form, which is more amenable for witness generation.

Unfortunately, DNF rewriting is not sufficient, because some mutual dependencies still remain among factors of conjunctions after DNF transformations. This means that we need to effectively push DNF transformation (as well as other operators) a step further in an unconventional fashion. The approach we have devised can be illustrated by the following example, where we focus on object schemas.

We use here JSON regular expressions (patterns), where $\hat{}$ matches the beginning of a string, $[\hat{}abc]$ matches any one character different from a, b and c, a dot . matches any character, \$ matches the end of the string, so that "$\hat{}\,a[\hat{}b]$." matches *acccccc* and *acc* but does not match *ac*, because the dot after the "$\hat{}\,a[\hat{}b]$" requires a third letter (carefully consider the dots in the patterns).

The expression below is a conjunction that we obtain by means of DNF transformations. We use the notation $\{\mathsf{Obj}, S_1, \ldots, S_n\}$ to denote a *group* of statements whose conjunction $\mathsf{Obj} \wedge S_1, \wedge \ldots \wedge S_n$ describes object values (we can also have array groups, etc.). Also note that $\mathbf{t}$ stands for the schema accepting any value.

$$\begin{array}{c}\{\mathsf{Obj}, \mathsf{props}(\hat{}\,a : S_1), \mathsf{props}(\hat{}\,.b : S_2),\\ \mathsf{pattReq}(\hat{}\,.d : \mathbf{t}), \mathsf{pattReq}(\hat{}\,a : S_3)\}\end{array}$$

A possible plausible witness generation strategy for this group would start considering pattReq constraints, but we need to keep into consideration possible interactions with other patterns in the object type, so we should first generate a witness for pattReq constraints, then checking whether props() are satisfied by the candidate witness, and if it is not the case, go back to pattReq and so on, by possibly infinite loops. To avoid this we rather manipulate the object group in order to be able to focus on subexpressions of the newly obtained object schema where, in some sense, all possible interactions are finitely enumerated, so that they can be dealt with separately.

Rather than providing the step-by-step process that produces this expansion, we show below the final result.

$$\begin{array}{l}\mathsf{props}(\hat{}\,a : S_1), \mathsf{props}(\hat{}\,.b : S_2) \rightarrow\\ \qquad \hat{}\,a[\hat{}b] : S_1, \hat{}\,ab : S_1 \wedge S_2, \hat{}\,[\hat{}a]b : S_2, \hat{}\,[\hat{}a][\hat{}b] : \mathbf{t}\\ \mathsf{pattReq}(\hat{}\,.d : \mathbf{t}) \rightarrow\\ \qquad \mathsf{orPattReq}(\hat{}\,ad : S_1 \wedge S_3, \hat{}\,ad : S_1 \wedge \neg S_3, \hat{}\,[\hat{}a]d : \mathbf{t})\\ \mathsf{pattReq}(\hat{}\,a : S_3) \rightarrow\\ \qquad \mathsf{orPattReq}(\hat{}\,ad : S_1 \wedge S_3, \hat{}\,a[\hat{}bd] : S_1 \wedge S_3,\\ \qquad\qquad\qquad\qquad \hat{}\,ab : S_1 \wedge S_2 \wedge S_3),\end{array}$$

In the props()-part the set $\{\hat{}\,a, \hat{}\,.b\}$ has been divided into three disjoint parts $\{\hat{}\,a[\hat{}b], \hat{}\,ab, \hat{}\,[\hat{}a]b\}$ by separating the intersection $\hat{}\,ab$ from the two original patterns, and the set is completed with $\hat{}\,[\hat{}a][\hat{}b] : \mathbf{t}$. Note that these new patterns can be obtained by means of standard techniques, thanks to the well-known closure properties of regular expressions.

The first request $\mathsf{pattReq}(\hat{}\,.d : \mathbf{t})$ is split into three different cases. The first $\hat{}\,ad : S_1 \wedge S_3$ is in common with the other orPattReq (an internal operator introduced to decompose pattReq into disjoint components), while the case $\hat{}\,ad : S_1 \wedge \neg S_3$ is internally and externally split, thanks to the $\neg S_3$ factor in the schema, and $\hat{}\,[\hat{}a]d$ is pattern-disjoint thanks to the initial $[\hat{}a]$. You can also observe that $\hat{}\,ad : S_1 \wedge S_3$ *internalizes* the requirement $\hat{}\,a : S_1$, the same holds for $\hat{}\,ad : S_1 \wedge \neg S_3$, while $\hat{}\,[\hat{}a]d$ only matches the trivial requirement, hence maintains its $\mathbf{t}$ schema.[1]

The second pattReq is split into three cases as well, in order to bring into view the intersection with the first pattReq, and in order to internalize the constraints of the props()-part.

This splitting effort is needed in order to be able to enumerate and try all the possible ways of satisfying a set of requests. For example, in this case the two orPattReq requests share the first component $\hat{}\,ad : S_1 \wedge S_3$, and contain two more components each, all of them mutually incompatible, hence having a structure $\mathsf{orPattReq}(a, b1, b2), \mathsf{orPattReq}(a, c1, c2)$. Hence, we know that there are exactly 5 ways of satisfying both: either by generating a single member that satisfies a, or by generating two members that satisfy, respectively, $(b1, c1)$, $(b1, c2)$, $(b2, c1)$, $(b2, c2)$, and our witness generation algorithm will try to pursue all, and only, these five approaches.

Even array groups obtained by DNF rewriting need preparation, by a different approach, which we cannot detail here, for

[1] As we have introduced not-schemas, notably $\neg S_3$, we re-apply not-elimination. For space reasons we do not delve into these aspects.

space reasons. Also, we have omitted how we deal with recursive definitions, both in not-elimination and witness generation.

This algorithm has an overall exponential complexity. However, we have designed techniques that make the problems tractable for many real-world schemas, and we are currently measuring their effectiveness.

4 DEMONSTRATION OVERVIEW

Our demo setup includes these datasets: we explore schemas from the JSON Schema Test Suite [2], a collection of small schemas that serve as unit tests for JSON Schema validators (and explore different operators), and real-world schemas from SchemaStore.org [5]. Naturally, our attendees may also formulate their own schemas.

We next describe the analysis for single schemas in more detail, and then remark on how attendees may also compare schemas.

Witness generation. Our tool is implemented as a Spring web application, with a Java backend. Our prototype does not yet support the operators `uniqueItems` and repeatedItems. Figure 1 shows a screenshot of the analysis of a single schema.

Typically, the user will first enter a JSON Schema document (or load one of the provided schemas), and then convert the schema (shown in the midsection of our screenshot) into our algebra (shown in the bottom section). Our algebra has been designed to be close to the original language, to be intuitive for practitioners. Yet different from the JSON Schema language, our algebra enjoys substitutability, that is, the semantics of an operator does not depend on its context, which eases manipulation.

The user may then choose to generate a first JSON witness. If the system finds no witness, it will alert the user that the schema is empty, otherwise, a witness is generated.

If there is a witness, the user can generate a further ("yet another") witness, that is different from all those previously seen. Alternatively, the user can edit the original JSON Schema, or directly the algebraic expression, and request that a new first witness is generated (disregarding witnesses already seen).

The schema designer can choose to convert back from the algebra to JSON Schema. Thus, the schema designer can interactively explore the semantics of a given schema, switch between the JSON Schema representation and the (often more compact) representation in our algebra, and iteratively revise the schema.

To allow interested demo attendees to inspect the internals of witness generation, as outlined in the previous section, our tool can also perform negation elimination on algebraic expressions. This feature would not be included in a tool targeted at end users.

Comparing schemas. Our tool offers a second screen (not shown here) where two schemas may be compared. Rather than computing a boolean answer to the question whether one schema subsumes the other, as done in state-of-the-art tools today [8], our tool can generate a witness that exemplifies a JSON document which is valid w.r.t. the one schema, but not the other.

Target audience. Our demo targets both the EDBT and the ICDT community. Attendees will become sensitive to the intricacies of working with the JSON Schema language, which caters to the ICDT community. Moreover, we point out original research questions that are of interest to the EDBT community, such as efficiency and scalability issues in dealing with real-world schemas, either due to the conditional semantics of the JSON Schema language, the interplay between negation and recursion (known to be difficult also in other areas of database research), and the sheer size of some real-world schemas (especially generated schemas, which can even take up hundreds of thousands of lines [8]).

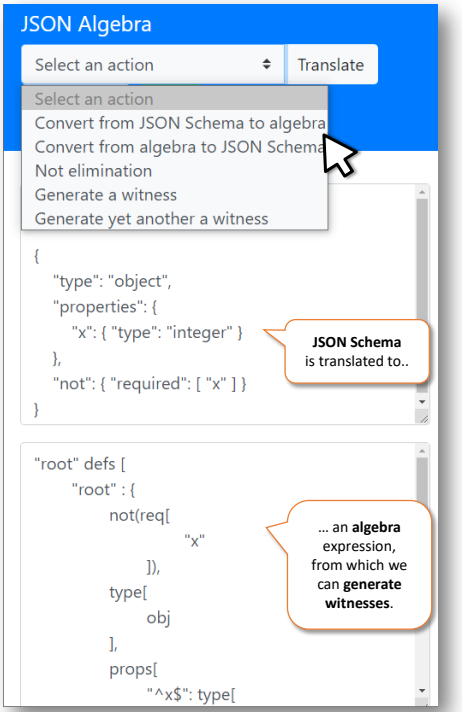


Figure 1: Screenshot: from JSON Schema to our algebra.

ACKNOWLEDGMENTS

Giorgio Ghelli's contribution has been funded by MIUR project PRIN 2017FTXR7S "IT-MaTTerS" (Methods and Tools for Trustworthy Smart Systems). Stefanie Scherzinger's contribution has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 385808805.

REFERENCES

[1] [n.d.]. JSON Schema – valid if object does *not* contain a particular property. Available at:https://stackoverflow.com/questions/30515253/json-schema-valid-if-object-does-not-contain-a-particular-property.
[2] [n.d.]. JSON Schema Test Suite. Available at:https://github.com/json-schema-org/JSON-Schema-Test-Suite.
[3] 2020. Apache Avro. http://avro.apache.org.
[4] 2020. Internet Engineering Task Force. Available at https://www.ietf.org.
[5] 2020. JSON Schema Schema. https://www.schemastore.org/json/.
[6] 2020. MongoDB. https://www.mongodb.com.
[7] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Not Elimination and Witness Generation for JSON Schema. In *Proc. BDA 2020.*
[8] Michael Fruth, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Challenges in Checking JSON Schema Containment over Evolving Real-World Schemas. In *Proc. EmpER.*
[9] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2020. Type Safety with JSON Subschema. arXiv:cs.PL/1911.12651
[10] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON Schema. In *Proc. WWW.* 263–273.
[11] Ion Stoica. 2020. Systems and ML: When the Sum is Greater than Its Parts. In *Proc. SIGMOD.*
[12] A. Wright, H. Andrews, and B. Hutton. 2019. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-handrews-json-schema-validation-02.* Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-handrews-json-schema-validation-02

covRew: a Python Toolkit for Pre-Processing Pipeline Rewriting Ensuring Coverage Constraint Satisfaction

Demonstration Paper

Chiara Accinelli, Barbara Catania, Giovanna Guerrini, Simone Minisi
DIBRIS - University of Genoa, Genoa - Italy
name.surname@dibris.unige.it

ABSTRACT

This demo presents covRew, a Python toolkit for rewriting slicing operations in pre-processing pipelines (i.e., pipelines to be executed before further tasks, such as data analytics and machine learning) so that the pipeline execution ensures that protected groups are adequately represented (i.e., *covered*) in the result. The toolkit includes: (i) an analyzer, which identifies candidate operations for rewriting; (ii) a rewriter, which transforms operations for ensuring coverage satisfaction with respect to user specified constraints; (iii) an impact evaluator, allowing the user to assess the impact of the rewriting on the obtained results.

1 INTRODUCTION

One of the main current challenges in data processing is the development of technological solutions satisfying non-discriminating requirements. Themes such as diversity, non-discrimination, fairness, protection of minorities, and transparency are increasingly crucial when processing and analyzing data.

Analytical pipelines processing real data are often very complex and various systems have been designed for supporting the user in the design and the execution of processing pipelines in a non-discriminating way. Among them, we recall: Fair-DAGs [11], an open-source library aiming at representing data processing pipelines in terms of a directed acyclic graph (DAG) and identifying distortions with respect to protected groups as the data flows through the pipeline; FairPrep [10], an environment for investigating the impact of fairness-enhancing interventions inside data processing pipelines; AI Fairness 360 [5], an open-source Python toolkit for algorithmic fairness, aimed at facilitating the transition of fairness-aware research algorithms to usage in an industrial setting and at providing a common framework to share and evaluate algorithms.

The complexity of data processing pipelines does not only depend on the used analytical or learning tasks but also on the types of pre-processing operations applied to input datasets for filtering, projecting (thus slicing), or merging together input objects. Indeed, it has been recognized that data pre-processing tasks can introduce bias at different levels [10]. As an example, classical data transformation operations, often defined in terms of Selection-Projection-Join (SPJ) operations over tabular data, can reduce the number of records related to some protected or disadvantaged groups, defined in terms of some sensitive attributes, even if such attributes are not directly used in the specification of the data transformation operation. As a consequence, some protected or disadvantaged categories can be under-represented in (*uncovered by*) the result of a transformation, possibly introducing bias in the following analytical steps.

As already recognized [3, 11], we believe that it is important to support the user in the design of non-discriminating data pre-processing tasks, with a special reference to data transformations defined in terms of slicing and merge operations. Additionally, following what stated in [9], we believe that such design can be improved by the usage of specific diversity or fairness-aware data transformations, where the idea is to *minimally* rewrite the transformation operation so that certain non-discrimination constraints are guaranteed to be satisfied in the transformation result. Through minimal rewriting, the revised process takes into account the original transformation goals and is traced for further processing, thus guaranteeing *transparency*.

Starting from these considerations, in our recent work [2], we designed an approach for minimally rewriting slicing and merge operations with the aim of satisfying specific *coverage constraints* [4, 6], guaranteeing that there are enough entries related to specific protected groups of interest in the result obtained by applying a given transformation, thus increasing diversity with the aim of limiting the introduction of bias during the next analytical steps. The problem we address is closely related to [3], where a constraint-based optimization approach is proposed for identifying a filter-based transformation generating a dataset satisfying a given input set of soft constraints. However, differently from [3], we consider the rewriting of transformations corresponding to SPJ queries with the aim of satisfying (hard) coverage constraints through a rewriting approach that can be easily integrated inside a pre-processing pipeline.

The aim of this demonstration is to showcase the techniques proposed in our recent work [2] by presenting covRew, a Python toolkit for rewriting data transformations specified inside pipelines described in Pandas [7], ensuring the satisfaction of a set of coverage constraints provided in input. The toolkit includes: (i) a *pipeline analyzer*, which identifies candidate operations for rewriting, (ii) a *pipeline rewriter*, which transforms operations that are selected by the user according to the input coverage constraints, and (iii) an *impact evaluator*, assessing the impact of the rewriting of the selected operations by comparing the result of the execution of the rewritten pipeline with that of the original one, according to the solution-based accuracy measures proposed in [1]. We showcase our toolkit in action with various scenarios on real-world datasets, demonstrating its usability in coverage constraint enforcing and the provided support for analyzing the impact of coverage-based rewriting. Notice that, though the approach supports merge operations and works on multiple datasets, in the demonstration, for the sake of simplicity, we will rely on a single dataset and will not consider merge operations. This way, indeed, the analysis of the impact of the proposed rewriting on the obtained results is clearer.

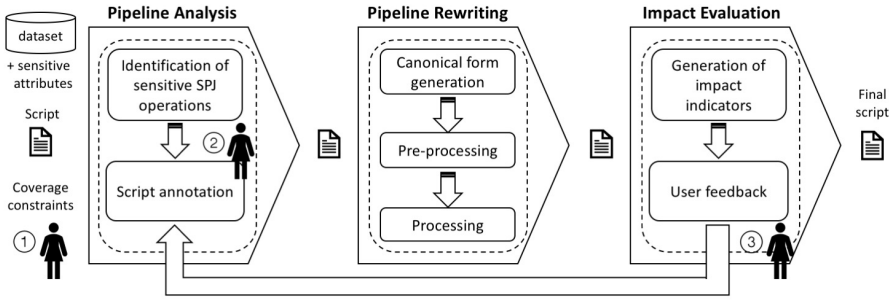


Figure 1: covRew flow

The remainder of the paper is structured as follows. In Section 2, we present the covRew architecture, illustrating the techniques underlying each step. In Section 3, we present the demonstration scenarios, with a special reference to the user interaction. Section 4 concludes and outlines some future work directions.

2 OVERVIEW

covRew is a Python toolkit focusing on the rewriting of data preprocessing pipelines, with the aim of satisfying specific coverage constraints on the results of slicing operations. In the following, we describe the characteristics of input data, processing pipelines, and the main covRew tasks. The covRew logical architecture is illustrated in Figure 1.

2.1 Input specification

Input data for covRew are: a dataset, with the related sensitive attribute specification, for the identification of protected groups; a processing pipeline represented as a Pandas script; a set of coverage constraints.

Dataset. covRew takes as input a tabular dataset I, represented as a Pandas Data Frame. We assume that some discrete valued attributes $S_1, ..., S_n$ of the input dataset are of particular concern since they allow the identification of protected groups and we call them *sensitive attributes*. Examples of sensitive attributes are the `gender` (with values in $\{female, male\}$) and the `race` (with values in, e.g., $\{asian, black, hispanic, white\}$).

Pipelines. covRew focuses on pre-processing pipelines represented in Pandas and in particular on Pandas data preparation tasks corresponding to *data slicing* operations.[1] The slicing operations we are interested in correspond to monotonic Select-Project (SP) queries over input tabular data that might alter the representation (i.e., the coverage) of specific groups of interests, defined in terms of sensitive attribute values. To this aim, we focus on SP queries that return, among the others, at least one sensitive attribute (called *sensitive SP operations or queries*). The sensitive attributes returned by an SP query Q are called *reference sensitive attributes* for Q. We further assume the user is satisfied by the specified transformations and she does not want to lose the obtained results through rewriting.

In the following, when needed, we denote Q by $Q\langle v_1, ..., v_d\rangle$ or $Q\langle\overline{v}\rangle$, $\overline{v} \equiv (v_1, ..., v_d)$, where $v_1, ..., v_d$ are the constant values appearing in the selection conditions contained in Q that, for the sake of simplicity, we assume to be numeric.[2]

Coverage constraints. Conditions over the number of entries belonging to a given protected group of interest returned as result by the execution of SP queries can be specified in terms of *coverage constraints* [4, 6]. Given an SP query Q with reference sensitive attributes $S_1, ..., S_n$, given a value s_i belonging to the domain of S_i, $i \in \{1, ..., n\}$, a coverage constraint with respect to S_i and s_i is denoted by $|Q \downarrow_{s_i}^{S_i}| \geq k_i$ and it is satisfied by Q over the input dataset I when $|\sigma_{S_i=s_i}(Q(I))| \geq k_i$ holds.

2.2 Pipeline analysis

In the first step, given the dataset, the list of sensitive attributes, the Pandas script, and the coverage constraints, covRew analyzes the script for identifying the sensitive SP queries and pre-annotates the script by highlighting them.

The user can then select, among the sensitive SP queries, those that, from her point of view, should be used for guaranteeing input coverage constraint satisfaction and, if needed for experimental purposes and for increasing system flexibility, she can change input coverage constraints. The output of this phase is a script annotated with information about the operations to be rewritten and related coverage constraints, if changed. By selecting different selective SP operations, during different covRew executions, the user can examine the impact of different coverage-based rewritings on the generation of the final result.

As an example, Figure 2 shows a Pandas script to be run on the US *Adult* Census database[3] containing information about 48,842 individuals from the 1994 U.S. census, taking `sex` as sensitive attribute and $|Q \downarrow_{female}^{sex}| \geq 100$ as coverage constraint, thus requesting that at least 100 females are returned by the selected SP operations. The pipeline analysis returns lines 7 and 17 as sensitive SP operations and the user can select one or both lines for the rewriting. In the following, we suppose line 17 is selected.

2.3 Pipeline rewriting

In the second step, covRew rewrites each selected sensitive SP query Q into another query Q', according to what presented in [2], so that Q' is the minimal query relaxing Q guaranteeing coverage constraint satisfaction when evaluated over the input dataset. More precisely, according to what stated in Section 2.1: (i) $Q' \equiv Q\langle\overline{u}\rangle$, thus it is obtained from Q without changing the original transformation goal; (ii) $Q \subseteq Q'$, thus the result of the original transformation is kept by the rewriting;[4] (iii) all coverage

[1] https://pandas.pydata.org/pandas-docs/stable/getting_started/intro_tutorials/03_subset_data.html

[2] We remark that the proposed approach can however be easily extended to deal with any other ordered domain.

[3] https://archive.ics.uci.edu/ml/datasets/census+income

[4] We remark that covRew can be easily extended by relaxing assumption (ii), in case query relaxation is not mandatory.

```
1  data = pd.read_csv('DEMO/dataset/adult.csv')
2
3  # projection
4  data  =  data[['capital_gain','age','education_num','sex',
   'capital_loss','hours_per_week','marital_status','label']]
5
6  # filtering
7  data = data.loc[(data['hours_per_week'] <= 70) &
   (data['capital_gain'] > 200)]
8
9  # OneHotEncoder considering marital_status
10 data = pd.concat([data[['age','education_num',
   'sex','capital_gain','capital_loss','hours_per_week'
   'label']],pd.get_dummies(data['marital_status'])], axis=1)
11
12 new_columns = list(data.columns)
13 new_columns.remove('label')
14 new_columns.remove('sex')
15
16 # filtering
17 data = data.loc[(data['education_num'] >= 15) &
   (data['hours_per_week'] > 40)]
18
19 # model
20 model = SVC()
21 model.fit(data[new_columns], data['label'])
```

Figure 2: Pandas script over the *Adult* dataset

constraints associated with Q are satisfied by $Q'(I)$. The coverage-based rewriting should be *optimal*, i.e.: (iv) there is no other query Q'' satisfying conditions (i), (ii), and (iii) such that $Q''(I) \subset Q'(I)$ (thus, Q' is the minimal query satisfying (i), (ii), and (iii)); (v) $Q' \equiv Q\langle\overline{u}\rangle$ is the closest query to $Q\langle\overline{v}\rangle$ according to the Euclidean distance between $\overline{v}$ and $\overline{u}$, satisfying (i), (ii), (iii), and (iv), in a normalized space in which the values for each dataset attribute are between 0 and 1, potentially increasing user satisfaction.

In order to compute the optimal coverage-based rewriting of an SP query $Q\langle\overline{v}\rangle$, given a set of coverage constraints CC and an instance I, we follow the approach presented in [2].

Canonical form generation. We first translate the selected SP queries into a *canonical form*, in which each selection condition containing operators ($>$, $\geq$, $=$) is translated into one or more equivalent conditions defined in terms of operator $<$. For example, any predicate of the form $A_i > v_i$ can be transformed into the predicate $-A_i < -v_i$. An optimal coverage-based rewriting of a canonical query is obtained from the original one by replacing one or more selection predicates $sel_i \equiv A_i < v_i$ with a *relaxed* predicate $sel'_i \equiv A_i < v'_i$ with $v'_i \geq v_i$. Relaxed queries generated through coverage-based rewriting starting from $Q\langle\overline{v}\rangle$, I, and CC have the form $Q\langle\overline{u}\rangle$, with $\overline{u} \geq \overline{v}$, and can be represented as points $\overline{u}$ in the d-dimensional space defined over the selection attributes, thus satisfying conditions (i) and (ii) of the reference problem.

Pre-processing. During the *pre-processing step*, we organize the reference space for the detection of a coverage-based rewriting of $Q\langle\overline{v}\rangle$ as a multi-dimensional grid. The grid has d axes, one for each selection attribute in $Q\langle\overline{v}\rangle$, and each axis is discretized into a fixed set of bins, by using the equi-depth binning approach, typical of histogram generation. Each cell in the resulting grid corresponds to a sensitive SP query containing $Q\langle\overline{v}\rangle$, in line with condition (ii) of the reference problem. The grid represents the search space for identifying the optimal coverage-based rewriting. The approach is *approximate* because a smaller coverage-based rewriting of the input query might exist but, if lying inside one grid cell, it cannot be discovered by the algorithm. Notice that the grid is computed starting from I and Q (CC is not used).

Processing. During the *processing step*, the multi-dimensional grid is visited starting from the cell corresponding to the input query, one cell after the other, at increasing distance from Q. For each cell ($\overline{u}$), we check whether the associated query $Q\langle\overline{u}\rangle$ is a coverage-based rewriting of $Q\langle\overline{v}\rangle$ by estimating the cardinality of $|Q(I)|$ and one cardinality for each coverage constraint. The properties of the grid and of the canonical form are considered for pruning cells that cannot contain the solution and for further improving the efficiency and the scalability of the process [2].

The processing step requires fast and accurate cardinality estimates. To make the processing more efficient and scalable, similarly to [8], we rely on estimators based on (uniform, independent, and without replacement) samples of the input dataset, dynamically constructed during the rewriting phase. covRew then allows the user to select two different rewriting modalities: *fast*, but potentially inaccurate, execution due to the usage of sample-based approaches for cardinality estimation (as a consequence, some constraints might not be satisfied when evaluated over the real dataset); *accurate*, but potentially slower execution, by detecting cardinalities in a precise way through query execution over the real dataset.

As a result of the pipeline rewriting step, covRew generates a rewritten script whose impact is evaluated in the final phase. In our example, line 17 of the original script is rewritten into: `data = data.loc[(data['education_num'] >= 14) & data['hours_per_week'] > 37)]`.

2.4 Impact evaluation

In the last phase, for each rewritten sensitive SP query, covRew shows many statistics useful for evaluating the impact of rewriting. In case the user is not satisfied by the rewriting, she can discard the changes and go back to the pipeline analysis pane, for changing her selections. More precisely, for each rewritten query, covRew returns (see Figure 3):

- the result of the rewriting of each original selection condition (in the example `data['education_num'] >= 14` and `data['hours_per_week'] > 37`), with information about the data distribution related to the selected attributes, before and after the rewriting;
- the maximum and the minimum approximation error due to the pre-processing, defined as the maximum and the minimum diagonal length of grid cells, normalized between 0 and 1, and the approximation error of the detected solution, corresponding to the grid-based accuracy proposed in [1] (0.04, 0.28, and 0.07 in Figure 3);
- the percentage of additional tuples returned, due to the rewriting, corresponding to the relaxation degree proposed in [1, 2] (66% in Figure 3);
- the Euclidean distance, in the normalized space, between the rewritten solution and the original one, corresponding to the proximity measure proposed in [1] (0.08 in Figure 3);
- the absolute and percentage distribution of protected groups in the result of the original query and of the rewritten one, (121 wrt to 22, 19.3 wrt 10.5 in Figure 3);
- information about the satisfaction of the coverage constraints on the result of the original query when evaluated on the input dataset (satisfied, in Figure 3).

The impact evaluation pane gives the user the opportunity to revise the annotation, if the accuracy is not satisfactory, and select the accurate execution if, due to the estimation error of the fast approach, some coverage constraints are not satisfied by the result of the rewritten queries over the input dataset. After this revision, the final script is returned to the user.

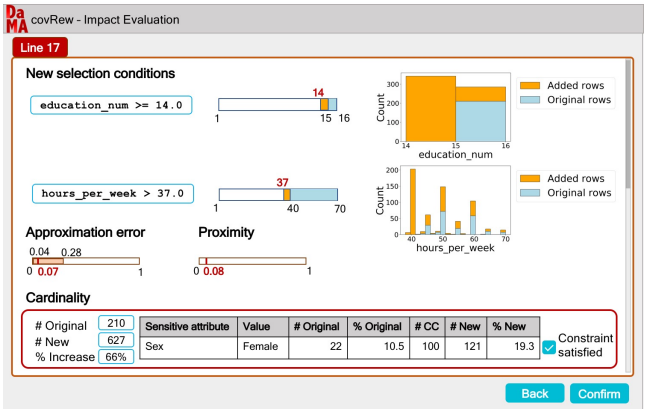


Figure 3: Impact evaluation pane

3 DEMONSTRATION SCENARIOS

In the demonstration of covRew we pursue three goals: (i) to ascertain the feasibility of rewriting slicing operators to ensure adequate protected group representation by means of coverage constraints; (ii) to explore the effects of such rewriting with respect to protected group membership on result accuracy; (iii) to compare executions of alternative rewritten pipelines sharing the same goal in terms of coverage, but using different rewriting approaches (efficiency vs accuracy). In the following, we discuss the foreseen user interactions and how these interactions realize the most relevant scenarios for our demonstration goals.

Datasets and sensitive attributes. As first operation, the user can select one dataset to work on ((1) in Figure 1). Two datasets are available: the already mentioned US *Adult* Census database. and the *Diabetes US*[5] dataset representing 10 years (1999-2008) of clinical care at 130 US hospitals and integrated delivery networks (100,000 instances). For each dataset, a short description of the dataset and the list of attributes is shown. The user is asked to select the sensitive attributes from this list for the task at hand and to define coverage constraints over them ((1) in Figure 1). For example pipelines, we select the `sex`/`gender` and `race` attributes for both datasets.

Input scripts. For each dataset, three different scripts are already available and will be proposed as a starting point. The user is allowed to freely modify the code as well as to enter her own code ((1) in Figure 1). A sample script is shown in Figure 2 and has been discussed in Section 2. The other scripts allow us to demonstrate different combinations of projection and filtering operations, with multiple conditions on different attributes, allowing to obtain a different ratio among different groups for sensitive attributes.

Script annotation and impact evaluation. In addition to selecting the input dataset and sensitive attributes, among those available, the user interacts with the toolkit in the pipeline analysis phase ((2) in Figure 1), when she can select the operations to rewrite, choose for each of them the execution type (fast rather than accurate) and, if needed, modify the input coverage constraints. The user can also provide a feedback after impact evaluation ((3) in Figure 1) when, by looking at the statistics about execution, she might want to reconsider some of the choices made in script annotation with reference to one or more slicing operations, thus producing a new annotated script.

[5]https://archive.ics.uci.edu/ml/datasets/Diabetes+130-US+hospitals+for+years+1999-2008

In the online demonstration, active user involvement for several simultaneous demonstration attendees will be fostered by using clickers/instant polling tools to select the input to provide to the system in the various steps.

Realized scenarios. The demonstration will be conducted in such a way to show, with reference to cases in which one or more operations are rewritten and one ore more coverage constraints are specified, different possible scenarios of interest, namely:

- the coverage constraints are satisfied and the user is fine with the rewriting;
- the coverage constraints are not satisfied on the input dataset and the user changes the type from fast to accurate;
- the user is not satisfied by the results, either because she deems the pre-processing approximation too high or because the rewriting has a too high impact on result accuracy, and decides to revise the choices she made, going back to the pipeline analysis pane.

In addition to the prefigured scenarios, the user will be given the opportunity to suggest modifications to the pipelines and even to submit her own script, on one of the reference datasets.

4 CONCLUSIONS

We have presented covRew, a user-friendly Python system for rewriting sensitive slicing operations that can lead to the violation of coverage constraints with respect to the protected groups of interests. In the demonstration, we illustrated the main covRew functionalities over predefined and user-specified pipelines. As future work, we plan to extend covRew with functionalities for automatically identifying sensitive operations to be rewritten with the highest accuracy. Further extensions concern the relaxation of the containment property between the original query and the rewritten one, a comparison of covRew with the approach proposed in [3], the integration with different types of fairness constraints, a graph-based representation of input scripts, similarly to [11], for simplifying pipeline analysis.

REFERENCES

[1] Chiara Accinelli, Barbara Catania, Giovanna Guerrini, and Simone Minisi. 2021. The Impact of Rewriting on Coverage Constraint Satisfaction. In *Proc. of the Workshops of the EDBT/ICDT 2021 Joint Conference.*

[2] Chiara Accinelli, Simone Minisi, and Barbara Catania. 2020. Coverage-based Rewriting for Data Preparation. In *Proc. of the Workshops of the EDBT/ICDT 2020 Joint Conference.*

[3] Dolan Antenucci and Michael J. Cafarella. 2018. Constraint-based Explanation and Repair of Filter-Based Transformations. *Proc. VLDB Endow.* 11, 9 (2018), 947–960.

[4] Abolfazl Asudeh, Zhongjun Jin, and H.V. Jagadish. 2019. Assessing and Remedying Coverage for a Given Dataset. In *Proc. of the 35th IEEE Int. Conf. on Data Engineering, ICDE 2019, 2019.* 554–565.

[5] Rachel K. E. Bellamy et al. 2019. AI Fairness 360: An Extensible Toolkit for Detecting and Mitigating Algorithmic Bias. *IBM J. Res. Dev.* 63, 4/5 (2019), 4:1–4:15.

[6] Yin Lin, Yifan Guan, Abolfazl Asudeh, and H. V. Jagadish. 2020. Identifying Insufficient Data Coverage in Databases with Multiple Relations. *Proc. VLDB Endow.* 13, 11 (2020), 2229–2242.

[7] Wes McKinney. 2002. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython.* O'Reilly Media, Inc.

[8] Chaitanya Mishra and Nick Koudas. 2009. Interactive Query Refinement. In *Proc. of the 12th Int. Conf. on Extending Database Technology, EDBT 2009,* 862–873.

[9] Babak Salimi, Bill Howe, and Dan Suciu. 2019. Data Management for Causal Algorithmic Fairness. *IEEE Data Eng. Bull.* 42, 3 (2019), 24–35.

[10] Sebastian Schelter, Yuxuan He, Jatin Khilnani, and Julia Stoyanovich. 2020. FairPrep: Promoting Data to a First-Class Citizen in Studies on Fairness-Enhancing Interventions. In *Proc. of the 23rd Int. Conf. on Extending Database Technology, EDBT 2020.* 395–398.

[11] Ke Yang, Biao Huang, Julia Stoyanovich, and Sebastian Schelter. 2020. Fairness-Aware Instrumentation of Preprocessing Pipelines for Machine Learning. In *Proc. of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2020.*

EasyBDI: Near Real-Time Data Analytics over Heterogeneous Data Sources

Bruno Silva
IEETA
University of Aveiro
Aveiro, Portugal
bsilva@ua.pt

José Moreira
DETI-IEETA
University of Aveiro
Aveiro, Portugal
jose.moreira@ua.pt

Rogério Luís de C. Costa
CIIC
Polytechnic of Leiria
Leiria, Portugal
rogerio.l.costa@ipleiria.pt

ABSTRACT

The large volume of currently available data creates several opportunities for sciences and industry, especially with the application of data analytics. But also raises challenges that make unfeasible the use of batch-based ETL processes. Indeed, near real-time data analytics is a requirement in several domains as an alternative to traditional data warehouses. In the last years, big data platforms have been developed to enable query execution over distributed data sources. However, they do not deal with subject-oriented analysis, do not provide data distribution transparency, or do not assist with schema mapping and integration.

In this demonstration, we present EasyBDI. It's a near real-time big data analytics prototype that enables users to run queries over heterogeneous data sources based on global logical abstractions created by the system and provides some usual concepts of data warehouse systems, like facts and dimensions. We use two motivating scenarios, one based on three years of real data on photovoltaic energy production and consumption, and the other based on the SSB+ benchmark. We will also present implementation challenges, issues, solutions, and insights.

KEYWORDS

Distribution transparency, data analytics, near real-time data warehousing

1 INTRODUCTION

For several years, data analytics has been based on large data warehouses. Such warehouses are mostly centralized databases whose data is periodically extracted from OLTP databases and load into the warehouse as part of an ETL (extract, transform, and load) process [10]. This traditional warehouse structure is not suitable for most of the current big data analytics environments. On the other hand, near real-time operations have become a requirement in several current IT contexts, like in IoT, where several sensors generate data streams and users need to process and analyze the most recent data [10].

This demonstration presents EasyBDI (*Easy Big Data Integration*), a prototype for logical integration of distributed and heterogeneous data sources (including NoSQL ones, like MongoDB, Kafka, and Redis) into a global database and global star schemas. The integration is logical, i.e., there is no materialized global database, and data source autonomy is maintained. Analytical queries specified over global star schemas are transformed and executed by the distributed and heterogeneous sources.

Building a global schema requires finding and matching syntactic and semantic similarities between the data structures of distinct local sources. This can be achieved either by looking at the structural organization of data (i.e., schema-based matching) or to the contents and meaning of data (i.e., instance-based matching) [7]. Also, each local element should be mapped into a global element. For instance, local structures identified as semantically identical in the schema matching process should be mapped into a single global entity. Partitioning of (logical) global structures across distinct databases should also be handled [6]. Creating a global schema over distributed databases is challenging, particularly in the context of NoSQL and heterogeneous databases. EasyBDI gets the data organization on participating sources and uses a combination of techniques to automatically propose a global schema, which can be fine-tuned by the users.

EasyBDI runs over a distributed query execution engine (Trino, formerly PrestoSQL [4, 8]) and adds some levels of abstraction, namely data location and fragmentation transparency, and specialized subject-oriented analysis. The system uses schema matching and integration techniques to automatically design a global model and allows users to build subject-oriented cubes over such model. Non-expert users may use drag-and-drop to submit analytical queries over global cubes, but advanced features (e.g. based on SQL language) are also available for experts.

Big data frameworks and polystore systems (e.g. Apache Drill [5], Presto [8], BigDAWG [3]) provide a unified query language that can be used to access distributed data. But big data frameworks commonly lack providing distribution transparency, while polystores are tightly integrated, managing all sources together, including in terms of data location and data replication [9]. Our system maintains source autonomy, uses a global schema to provide distribution transparency (location, replication, and fragmentation), is extensible, and supports a wide range of data sources.

In the demonstration, two scenarios will be made available for participants. The first one is based on more than 3 years of real data on photovoltaic panel production/consumption in Sydney, Australia, and nearby areas. The second uses the SSB+ benchmark [2], which contains persistent and streaming data on retail store's sales, deliveries and popularity in social media.

Participants will understand how EasyBDI deals with some key challenges, like how to (i) explore the local data models to identify entities of the global model that are partitioned across multiple data sources, (ii) implement the automatic schema matching, mapping, and integration procedures to support the users in designing global models for a large number objects and data sources, and (iii) execute queries on a high-level star schema model abstracting several distributed, heterogeneous and autonomous data sources. They will also see the global SQL queries generated by EasyBDI and their translation into queries to the local sources. Implementation challenges and issues, adopted solutions and insights will be discussed, making this demonstration helpful to researchers and practitioners.

2 EASYBDI PROTOTYPE

EasyBDI is a framework for near real-time analytics that provides logical integration of multiple data sources while also enabling the creation of star schemas over global (logical) entities. Local databases are assumed to be autonomous (operate independently) and heterogeneous (in terms of data models and query languages, e.g., relational, graph and document databases, or semi-structured and unstructured data sources).

Data sources are accessible through a distributed query engine (Trino). The role of EasyBDI is to add additional levels of transparency, namely, location and fragmentation transparency, to allow building global schemas providing unified and high-level representations of the data sources, and to enable the execution of analytical queries by subject-experts. The architecture and the main components of EasyBDI are depicted in Figure 1.

API communication handler This component handles the interaction with the external systems: Trino and SQLite.

The *Distributed Query Execution Engine Interface* uses Java and JDBC, and implements all the logic regarding the integration of EasyBDI with Trino. Even though Trino provides a single representation of the underlying data, queries to Trino must contain the data source identifier for each data fragment. Thus, this component must also convert the queries specified using a global schema (see Database Integration below) into queries supported by Trino. This requires adding data source identifiers, as well as union and join operators, according to the mapping used between the global and local schemas.

The *Schema Metadata Storage* is responsible for the storage and management of the metadata of all schemas: local schema view, global schema, and star schema. Currently, these (meta)data are stored in an SQLite database.

Configuration Manager This component is responsible for generating the configuration files that allow Trino to communicate with the data sources (e.g., data source type, data source URL, username, password, and other parameters that depend on the data source type). It also creates local schema views. The procedure consists of iterating each data source and retrieving (meta)data about their schemas, namely, tables and columns information (or the equivalent concepts depending on the data source), using Trino commands. These (meta)data are stored in an SQLite database through the *Metadata Storage Interface*.

Database integration This layer deals with the creation of the global schema. A global schema contains a set of global tables, each containing a mapping to logically related data fragments. The integration is logical, i.e., the global schema is entirely virtual and not materialized. The main tasks to build a global schema are *schema matching*, *schema integration* and *schema mapping*.

The *schema matching* defines a mapping of concepts in a schema with concepts in another schema. EasyBDI uses a schema-based method with linguistic and constraint-based criteria [6]. The algorithm starts by finding tables with similar names using the Levenshtein distance. For each pair of matching tables, EasyBDI does columns matching using the Levenshtein distance to compare names and a similarity measure to compare data types.

The *schema integration* defines the global tables and their columns, using the correspondences found in the schema matching. EasyBDI uses the stepwise binary integration method [6].

The *schema mapping* defines how to combine data from one or more local data sources (data fragments) into a single global table while keeping consistency and semantic coherence. EasyBDI

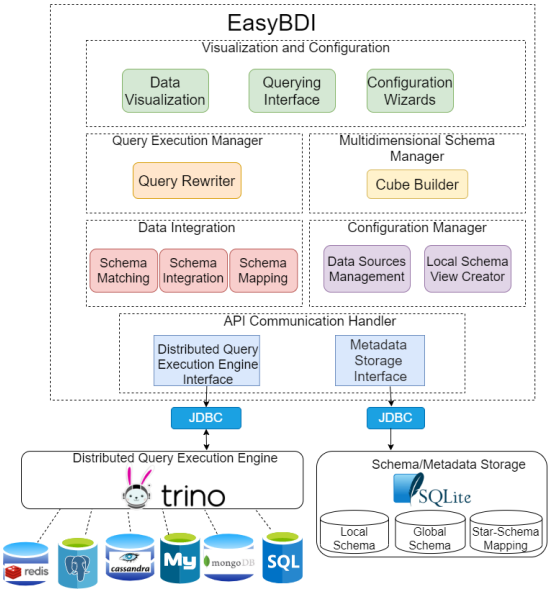


Figure 1: Main Components of EasyBDI architecture

analyzes the matching between global and local entities, and identifies the type of partitioning (horizontal, vertical, or none) used. If a global table corresponds to only one local entity, then there is no partitioning. If a global table has a correspondence with several local tables, and the number of matching columns and their data types in all tables are the same, then the local tables are considered horizontally partitioned. The current implementation of EasyBDI uses foreign key-primary key relationships to find whether two or more tables are vertically partitioned, but this is only feasible when it is possible to get constraint information from the catalog of the data sources.

The methods presented above are automatic and may lead to incomplete or semantically incorrect results. Thus, EasyBDI allows users to review and edit the global schema generated automatically (e.g., removing replicated data sources) using an intuitive GUI interface.

Multidimensional Schema Manager This component allows the design of data cubes (star schemas) and the use of abstractions like facts and dimensions to perform analytical queries so that users focus on data analysis rather than technical details regarding data organization. Data cubes are built over the global schema,i.e., fact and dimensions tables are based on global entities. A start query is basically a join between a fact table and some dimensions, possibly with filters and aggregations.

Query Execution Manager This component rewrites the queries on the global schema into the queries submitted to Trino to get data from the local data sources. It handles vertical and horizontal data partitioning. Queries on global tables are automatically translated into queries on local tables using union and join operations of data fragments. The framework can handle multiple aggregations and joins at the same time. The queries that merge partitioned data are written as nested queries. An outer query contains the operators specified by the user (e.g., filters and aggregations) and other implicit joins needed between the facts table and dimensions. It is also possible to deal with complex operations such as pivoting and unpivoting data.

Figure 2 exemplifies the query submission process, which starts with a user interacting with the interface and issuing a

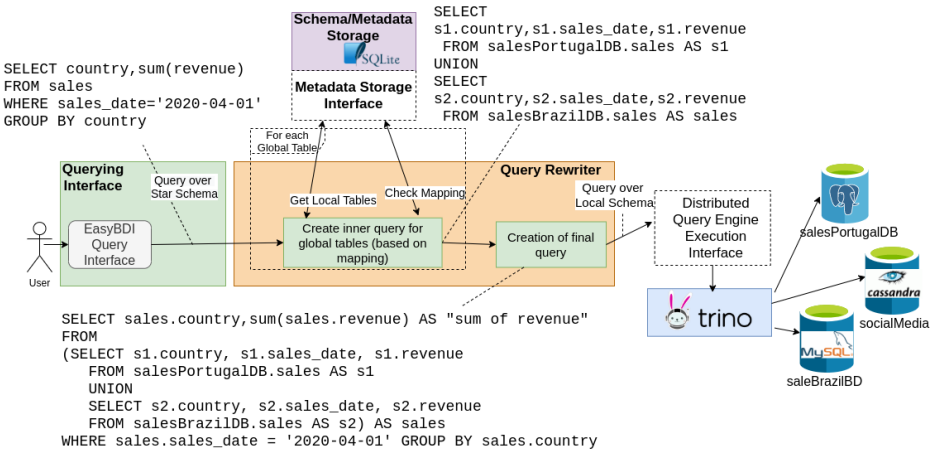


Figure 2: Query transformation - from an initial query over global schema to a query over local schemas

query over the star schema. Then the system generates an SQL query based on the global schema, which is translated into a query on the local schema and submitted to Trino. Notice that users do not need to know any query language: EasyBDI provides an intuitive interface and analytical queries are specified using drag-and-drop. Finally, Trino prepares and sends the queries that must be executed by the local data sources. Thus, users can create queries using global names and cubes and do not need to know about the format, organization and distribution of the data, nor a query language.

The *Query Execution Manager* also transforms the results provided by Trino according to the global schema and sends them to the visualization layer.

Visualization and Configuration This layer comprises several tools related to querying and configuration, including wizards to provide guided operations to users, like the cube builder, query builder, global schema editor, and data source configuration.

3 DEMONSTRATION

This demonstration has two case studies to highlight different features of EasyBDI. It will cover the selection of data sources, automatic generation of a global schema, creation of a star schema and execution of drag-and-drop (and user-edited) analytic queries.

Case study 1 uses data on energy production and consumption of 300 homes equipped with photovoltaic panels in Sydney over the span of 3 years. The data are available in three CSV files. The lines represent the customers, generator capacity, dates and type of consumption or production (GC, CL, and GG), and the columns represent the values recorded every 30 minutes (Figure 3). This case is interesting because the organization of the data is far from a typical data organization in relational databases.

We consider that the data on the customers' location (postal codes related data) are in a PostgreSQL database and the temporal data are in a MySQL database, just for experimentation purposes.

Customer	Generator Capacity	Postcode	Consumption Category	date	00:30	01:00	01:30	...	23:30	00:00
1	3.78	2076	GC	01/jul/10	0.303	0.471	0.083	...	0.078	0.125
1	3.78	2076	CL	01/jul/10	1.25	1,244	1.256	...	0	1.075
1	3.78	2076	GG	01/jul/10	0	0	0	...	0	0
1	3.78	2076	GC	02/jul/10	0.116	0.346	0.122	...	0.12	0.111

Figure 3: Sample of a CSV file with photovoltaic data

After the automatic generation of the global schema, we will manually edit it to show some advanced features. In particular, we will show how to use virtual tables and user-defined commands to unpivot the data in the CSV files, how to specify constraints and change the data types of global schema columns, and how to create a mapping between the global schema and the data sources using virtual tables (Figure 4).

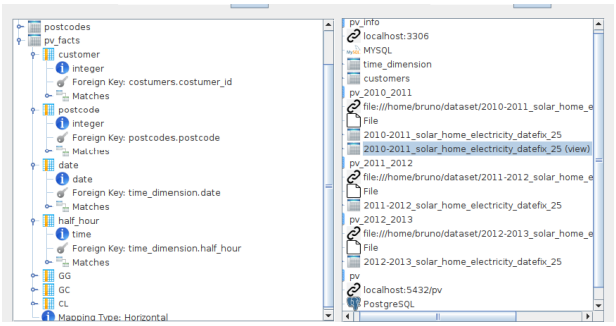


Figure 4: Global schema (left panel) and local schema views (right panel) for the photovoltaic datasets

Then, a star schema will be created with a fact table (pv_facts), three measures (GC, CL and GG) and three dimensions (customer_dim, time_dim and postalcodes_dim). We will show how to run queries on the global schema using only drag-and-drop how to edit the commands manually (Figure 5). We will also illustrate the transformation of queries on a global schema (Listing 1) into queries on the local schema views that must be executed by Trino (Listing 2). The "<user query>" in Listing 2 denotes the query used to transform the data structure depicted in Figure 3 into a virtual table and is omitted because of its size.

Listing 1: Code generated for the global query depicted in Figure 5 (top) (datatype casting operators were removed).

```
SELECT    c.customer_id, t.year,
          SUM(pv.GG) AS "SUM_of_GG"
FROM      customers c, time_dimension t, pv
WHERE     ( t.half_hour = pv.half_hour
AND         c.customer_id = pv.customer
AND         t.date = pv.date )
GROUP BY ( t.year, c.customer_id)
```

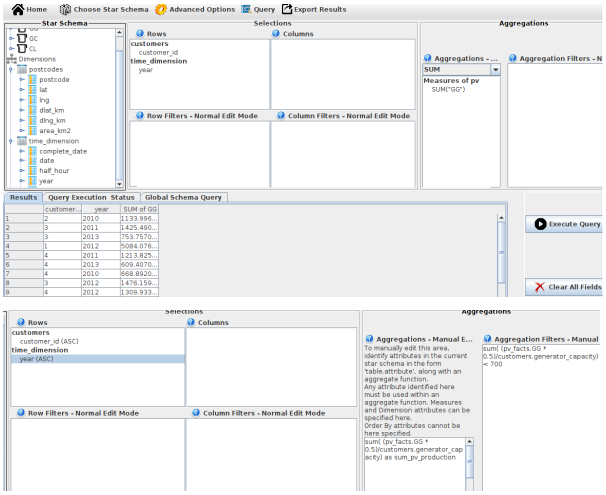


Figure 5: A global query made with drag-and-drop (top) and a global query edited manually (bottom)

Listing 2: Translation of the code in Listing 1 into a query over the local schema views (the name of the data source "mysql_localhost_3306_time" is abbreviated to "mysql").

```
SELECT customers.customer_id, time_dimension.year,
       SUM(GG) AS "SUM_of_GG"
FROM
  (SELECT mysql.pv_schema.customers.customer_id
   FROM mysql.pv_schema.customers) AS customers,
  (SELECT mysql.pv_schema.time_dimension.half_hour,
   mysql.pv_schema.time_dimension.year,
   mysql.pv_schema.time_dimension.date
   FROM mysql.pv_schema.time_dimension) AS
   time_dimension,
SELECT customer, date, half_hour, GG FROM
   (<user query> UNION <user query>
   UNION <user query>) AS pv
WHERE (time_dimension.half_hour = pv.half_hour
   AND customers.customer_id = pv.customer
   AND time_dimension.dated = pv.date)
GROUP BY(time_dimension.year, customers.customer_id)
```

Case study 2 uses the SBB+ benchmark [2] and represents a more conventional scenario in a big data environment. The SSB+ data model has two star schemas, one for batch Online Analytical Processing (OLAP) and the other for streaming OLAP. The batch OLAP is an adaptation of the schema proposed in TPC-H. The streaming OLAP is based on social media data and represents the popularity of retail stores (and their sales and deliveries).The data for OLAP batch storage are stored in Hive, while the facts data for streaming OLAP are stored in Cassandra (a NoSQL database that uses a wide-column store model), and conceptual relationships exist between data stored in both systems, as represented in Figure 6. We use the code available in [1] to populate the data sources.

In this case, EasyBDI's automatic matching, integration and mapping was mostly correct. Only a few manual edits are needed and no user-defined commands are necessary. We also show how to create the star schemas for batch OLAP and streaming OLAP and how to execute queries. SSB+ has a listing of analytical queries that we used to test the functionality of EasyBDI and we also use some of these queries in this demonstration. Regarding EasyBDI's performance, the overhead associated with query

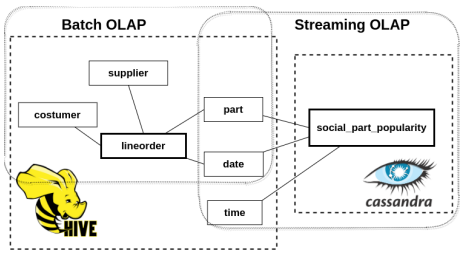


Figure 6: Case study 2: batch and streaming OLAP – overview of local schemas

generation is small (≈ 1% of query execution time). Query execution time depends mostly on query complexity, data sources' efficiency, and on the resources available for Trino.

4 SUMMARY

In this work, we present EasyBDI, a framework for near real-time data analytics that uses logical data integration to provide a high-level abstraction of data distribution and heterogeneity, while keeping the autonomy of the data sources. Automatic schema matching and configuration wizards are used to support the addition of new data sources. Multidimensional data organization is used to enable query analytics by subject-experts. We present two scenarios, one based on real data and the other on the use of a benchmark that simulates sales data and social media data. We also present several implementation issues, discussing adopted solutions that may be helpful to researchers and practitioners. EasyBDI is publicly available on Github *https://github.com/bsilva3/EasyBDI*.

ACKNOWLEDGMENTS

This work is partially funded by National Funds through the FCT (Foundation for Science and Technology) in the context of the projects UIDB/04524/2020, UIDB/00127/2020 and POCI-01-0247-FEDER-024541.

REFERENCES

[1] Carlos Costa. 2019. Big Data Benchmarks. https://github.com/epilif1017a/bigdatabenchmarks Accessed = 2021-02-01.
[2] Carlos Costa and Maribel Yasmina Santos. 2018. Evaluating several design patterns and trends in big data warehousing systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10816 LNCS. Springer Verlag, 459–473. https://doi.org/10.1007/978-3-319-91563-0_28
[3] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. 2015. The BigDAWG Polystore System. *SIGMOD Rec.* 44, 2 (Aug. 2015), 11–16.
[4] Trino Software Foundation. 2021. Trino - Distributed SQL Query Engine for Big Data. https://trino.io/ Accessed = 2021-02-01.
[5] Michael Hausenblas and Jacques Nadeau. 2013. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *Big data* 1, 2 (2013), 100–104.
[6] M. Tamer Özsu and Patrick Valduriez. 2020. *Principles of distributed database systems, fourth edition.* Springer. 1–674 pages.
[7] Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *VLDB Journal* 10, 4 (dec 2001), 334–350.
[8] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on everything. *Proceedings - International Conference on Data Engineering* 2019-April (2019), 1802–1813.
[9] Dimitris Stripelis, Chrysovalantis Anastasiou, and José Luis Ambite. 2018. Extending Apache Spark with a Mediation Layer. In *Proceedings of the International Workshop on Semantic Big Data* (Houston, TX, USA) *(SBD'18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages.
[10] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. 2017. Enabling query processing across heterogeneous data models: A survey. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*. IEEE Computer Society, 3211–3220.

Very Short Primer on Blockchain Technology for Database Researchers

Zsolt István
IT University of Copenhagen, Denmark
zsis@itu.dk

ABSTRACT

Blockchain is an emerging technology, considered increasingly often beyond the cryptocurrency world for business-to-business use-cases. In contrast to public blockchains such as Bitcoin, that are open systems in which anyone can participate, in business-to-business scenarios the membership of the service is controlled (permissioned blockchain). This permits the use of Byzantine fault tolerant (BFT) consensus protocols at the core of the service to establish a total order of transactions, instead of the more expensive Proof-of-Work-based consensus protocols. Permissioned blockchains typically set out to solve problems in the space where databases have traditionally resided, with the main difference being that the former decentralizes trust. There are numerous research proposals in the intersection of databases and blockchains. Sadly, there are still many misconceptions about this technology which leads to confusion in the community. The main goal of this primer is to give an overview of the relevant topics and provide pointers for further reading.

1 BLOCKCHAIN BASICS

Blockchains have entered the "spotlight" after the seminal Bitcoin paper published by Nakamoto [11] more than a decade ago. Even though many think of Blockchains as being part of the data management field, the Nakamoto paper was addressing a financial issue: its goal was to provide a decentralized currency that does not require trust in a central authority for its functioning. It has been only later that discussions emerged about "transaction processing", that is, smart contracts on top of blockchains, perhaps most notably, in the Ethereum White Paper [4].

In the years since their proposal, Blockchains have lived through a hype and this resulted in a dizzying number of different systems. At their core, however, all blockchains are quite similar. They aim to *decentralize trust* and are composed of two parts: *1) a verifiable data structure* that allows participants to determine whether the data in the blockchain has been tampered with and *2) a consensus algorithm* that defines how participants can add new data to the data structure. The choice for data structure is typically an append-only log with cryptographic hashes linking entries (i.e., blocks, see Figure 1). Nonetheless, there are also blockchains, such as IOTA, that order data into a directed acyclic graph (DAG) [10] instead. Put in database terms, we can think of the former as establishing global total order on all transactions and keeping a single logical shard of the data, and the latter as data sharding with total order within a shard and infrequent cross-shard transactions.

The choice of the consensus algorithm is determined by the assumptions the blockchain makes on the trustworthiness of third parties and the participation model. We can split blockchains into two categories based on who is allowed to participate in the consensus algorithm and, as a result, add data to the data structure: First, *permissionless blockchains* where anyone is allowed to participate and, second, *permissioned blockchains* where only selected entities can participate (this also assumes that the nodes in the system have verifiable identities, which is not the case in permissionless blockchains).

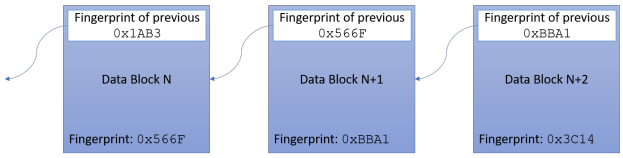


Figure 1: At the core of all blockchains is a verifiable data structure, typically in the form of an append-only log. Cryptographic hashes (fingerprints) included in blocks recursively tie them to the previous blocks.

Permissionless blockchains are, by definition, *public*, since anyone can access the data structure and read all previous transactions. Permissioned blockchains can be either *private*, in which case the governing consortium or members of the blockchain restrict access based on internal rules (e.g., business partners in specific areas), or they can be *public*, in which case anyone fulfilling some non-restrictive condition can join (e.g., Alastria[1] implements a country-wide blockchain network that relies on national tax register numbers as a pre-condition to joining). It is important to note that, while permissionless blockchains are typically used for dealing with fully virtual assets, permissioned blockchains are more similar to databases, in that the data they store typically represents real-world assets and events.

What is there for database researchers to explore? In the permissionless blockchain space most current and future challenges are related to scalability of the network, to economic incentives, as well as, to consensus algorithms with better guarantees for tolerating malicious actors. In terms of data management and transaction processing, there is no clear need for inventing new approaches because these systems can benefit from already existing best practices. In contrast, permissioned blockchains are considered for use-cases which are much closer to the database community, such as supply-chain management, banking or notaries, and therefore face challenges closer in nature to the database community. As we will discuss in the last section, there is significant overlap in database and blockchain research, with many database ideas/techniques being well applicable in the blockchain space. The fundamental difference, however, between permissioned blockchains and traditional distributed databases is the decentralization of trust in the former, that is, the assumption that not all nodes belong to the same enterprise and the ability to function even if not all members of a consortium are trusting each other. From a database research perspective, hence, most

[1] https://alastria.io/en/

of the interesting future work is in understanding how the performance of transaction processing (and perhaps analytics) can be increased while benefiting from the trust decentralization of blockchains. Additionally, the standardization of smart contracts and data models, as well as their integration with traditional databases is an open challenge.

2 PERMISSIONLESS BLOCKCHAINS

In permissionless blockchains there are no node identities and, as a result, a consensus mechanism is required that cannot be subverted by creating a large number of "fake" nodes to achieve majority in the system (Sybill attack). Therefore, Bitcoin and many other cryptocurrencies use *Proof of Work* (PoW) consensus, that requires participants to the consensus to solve a cryptographic puzzle before they can add to the log. The puzzle consists of finding a hash value that fulfills a specific condition, for instance, a number of leading zeros. The input to the hashing is the block that the participant would like to record on the blockchain. Inside the block are 1) transactions collected from clients of the system and other participants, 2) the cryptographic hash (signature) of the previous block in the blockchain and 3) a field holding a random number. The randomness is added to the block before being hashed, and unless the resulting hash fulfills the condition, the last step is repeated (this is the process of *mining*). Finally, when a miner discovers a combination for which the condition holds, it appends it to the blockchain by broadcasting it to all other participants. These can *validate* whether the added block is legitimate simply by computing its cryptographic hash and checking whether this hash fulfills the condition[2]. Hence, even though mining is very expensive, clearly limiting the throughput of the system, validation itself is cheap. In exchange for "finding" the next block of the blockchain, the miner will receive a payment in the underlying cryptocurrency, typically deducted as fees from the transactions in the just-appended block.

While PoW consensus has benefits in that it limits participants' power in the blockchain to their relative compute power (which is much more costly to increase than creating new IP addresses for instance), it has also many drawbacks. Apart from the obvious energy efficiency concerns resulting from the repeated hashing computations and the issue that all miners are fundamentally competing with each other for the next block, it has an important implication on *transaction finality*. In systems relying on PoW, competing miners could create forks in the log, working towards two "alternate realities". In practice, blocks are considered committed once enough additional blocks (e.g., 5) have been added after them. The expectation is that, at that point, the economic incentives are driving all participants to continue building on the longest subchain of the system. Nonetheless, transactions never reach *finality* because there is a small probability that any block, no matter how old, could have been forked and it does not, in fact, lie on the longest subchain.

It has to be noted that the common practice of composing *blocks* from a large number of transactions is rooted in the space of PoW blockchains in an effort to amortize the exuberant cost of mining, as well as, to simplify global broadcasts. Overall, the idea of a blockchain and its underlying mechanisms could just as well work for appending single transactions into a tamper-proof log, albeit, with much less efficiency for PoW consensus-based blockchains, such as Bitcoin.

To overcome the fundamental efficiency issues of PoW, other forms of consensus are becoming wide-spread, most notably, variations of Proof of Stake (PoS), e.g., in Tezos, Peercoin, and in newer iterations of Ethereum. PoS consensus is, at its core, a majority-based consensus algorithm that requires the voting participants to stake part of their cryptocurrency holding behind each consensus round. In case irregularities happen, e.g., forks, or conflicts, they are liable to lose their stake. This creates a strong economic incentive for participants to adhere to the rules.

In addition to PoS, there are other exciting proposals, such as Proof of Storage, Proof of Elapsed Time and Proof of Personhood, to name only a few (see more in this survey [14]). In addition to consensus protocols that provide a "proof" of owning some information or property, there are other proposals that embrace stochastic behavior and can even tolerate 51% attacks temporarily [13].

Even though non-PoW consensus solves many of the efficiency issues of PoW blockchains and, in principle, allows for lower latencies, these blockchains still typically suffer from the lack of finality in transactions: their throughput and latency are not determined only by the choice of underlying consensus protocol but also by the economic assumptions the blockchain makes.

3 PERMISSIONED BLOCKCHAINS

Permissioned Blockchains, such as Hyperledger Fabric [2], Corda R3 [3] and IOTA [12], introduce the requirement for a mechanism to associate identities with participants of the blockchain. Identities could be either issued by a trusted third party or by a consortium of the blockchain nodes themselves. Identities allow the use of more traditional, and significantly cheaper, forms of consensus algorithms, typically those from the family of Byzantine Fault Tolerant (BFT) algorithms (e.g., PBFT [5]), to append to the shared data structure. Using BFT consensus has the important benefit that transactions can be committed with *finality*. As a result, these permissioned systems can reach latencies and throughputs more similar to those of traditional databases. It has to be noted, however, that many of these systems run in widely geodistributed setups without the availability of dedicated, high bandwidth, networking, resulting in lower performance than what we typically see in RDBMSs.

In addition to the limitations imposed by networking bottlenecks, many permissioned blockchains inherit other limitations from the PoW blockchain space. They often rely, for instance, on batching a large number of transactions into blocks, which unsurprisingly results in artificially high latencies. These limitations, however, are not fundamental. We have demonstrated, as an example, that by rethinking the block-based processing of Hyperledger Fabric [9], its latency can be lowered from the hundreds of milliseconds to the millisecond range without negatively impacting its throughput or requiring significant changes to its code base. Therefore, it is reasonable to assume that permissioned blockchains that are being deployed in production systems will eventually "shed" the most obvious inefficiencies.

Given the decades-long of research in distributed consensus and fault tolerant systems, it is reasonable to predict that the work of database researchers will not necessarily be most useful in improving consensus but, instead, in enhancing the data management and data processing aspects of these systems. For instance it is an open challenge how to make Smart Contract

[2]Depending on the nature of the transactions in a block, of course, additional checks might be required that ensure that the underlying state remains consistent at all times.

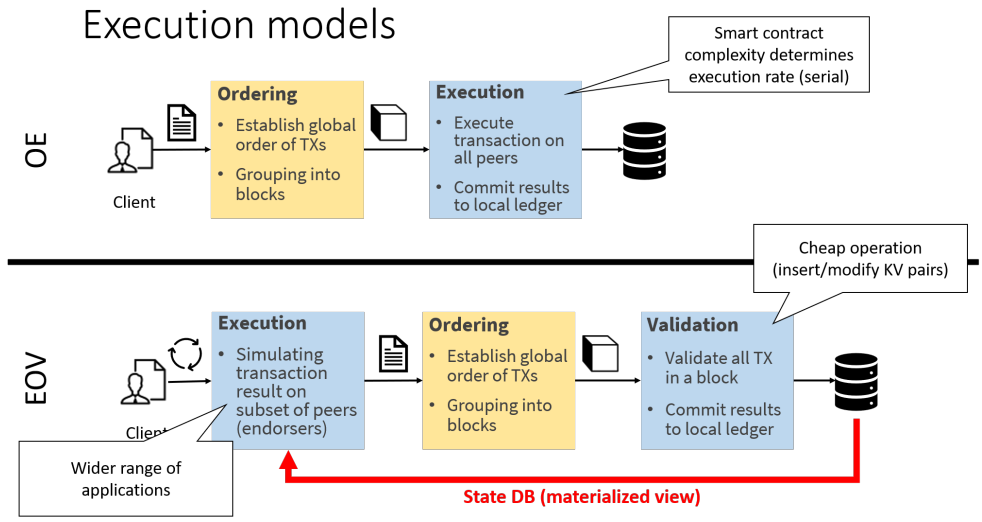


Figure 2: Blockchains can be classified into two groups depending on their smart contract execution model (nomenclature might differ across systems)

execution more efficient and integrated with RDBMs, which, in some cases, might have different schemas at different companies.

4 SMART CONTRACT EXECUTION

Most practical solutions for Smart Contracts (e.g., in Ethereum, Tezos, Fabric, etc.) treat the "contents" of the blockchain as a multi-versioned key-value store. As such, it is possible to think of smart contracts as small programs that can only interact with a key-value interface.

Blockchains adopt one of two execution models: *Order-Execute* (OE), which is similar to active replication in database terminology, and *Execute-Order-Validate* (EOV), which can be thought of as passive replication. As shown in Figure 2, the OE model first establishes the total order of transactions (that is, the order of smart contract invocations), then broadcasts these to all participants, each participant executing the transactions locally, typically in a serial manner. For this approach to work, smart contracts cannot include non-deterministic operations, otherwise the results could diverge. Blockchains using the OE model ensure this by relying on carefully designed DSLs for writing smart contracts.

Systems implementing the EOV model start by simulating (also called executing or endorsing) smart contracts on a subset of the nodes, chosen by a user-defined policy. The results of these executions are recorded in a terms of read/write modifications they would perform on the key-value store (state DB). Once the client receives enough simulation results, and these are all identical, it can submit the transaction for ordering (that is, the R/W modifications and proofs of execution). Once the order of transactions has been established, all participants will receive the list of transactions to record in their ledgers. At this point, the smart contracts are not re-executed but instead their read/write modifications are made directly on the ledger. The benefit of this solution is that it allows executing a large number of smart contracts in parallel but, since all simulations happen on a "view" of the ledger state that can change by the time validation happens, transactions can fail in the validation phase due to data staleness in the state DB in the first phase. In the EOV model it is feasible to use general purpose programming languages, such as Go or Java, to write smart contracts since (most) non-deterministic behavior will be detected in the simulation phase.

5 RECENT RELATED WORK IN DATABASES

In the following, we present a handful of related works published within the database community targeting shortcomings of permissioned blockchains mentioned above (the list is not exhaustive and is intended as a sample of the space).

In an effort to increase transaction processing throughput in the EOV model, Sharma et al. [16], show that by relying on database techniques for concurrency control, it is possible to reorder transactions within a block during the ordering step (i.e., when establishing their total order) in a way that minimizes the number of failing transactions due to R/W conflicts. They implemented their prototype on Fabric and it is one example of how ideas from the database world can be used to improve existing blockchain platforms without fundamental redesigns. Other examples include FastFabric [7], that brings various optimizations, inspired by databases, to Fabric which result in an unprecedented 20,000 ops/s throughput.

There are also proposals which aim to increase throughput and lower latencies by implementing sharding at different levels in permissioned blockchains. CAPER [1], for instance, allows multiple applications to share one blockchain and leverages the fact that they operate on disjoint parts of the dataset to increase the overall throughput. This is achieved by separating the ordering of operations inside an application from that of ordering across applications. In a similar vein, ResilientDB [8] proposes a permissioned blockchain that incorporates a hierarchical consensus protocol design that relies on locality, both in terms of dataset and physical proximity of the nodes, to boost performance.

Other lines of work treat blockchains as a fault-tolerant and highly available storage layer and build traditional database capabilities on top, e.g., as in BlockchainDB [6]. This direction of research provides one possible answer the question of how to bridge the gap between SQL-based processing (and the amassed expertise in this space by developers) and the fairly exotic field of smart contracts. Other work, e.g., ChainifyDB [15], explores a similar

question and provides a different possible answer. Instead of replacing the storage engines of databases, ChainifyDB re-designs the distributed transaction processing protocol and utilizes a blockchain for BFT fault tolerance and transparency/auditing for both local and distributed transactions.

ABOUT THE AUTHOR

Zsolt István is an Associate Professor at the IT University of Copenhagen, working in the area of databases, distributed systems, and FPGA programming. Earlier, he was an Assistant Research Professor at the IMDEA Software Institute in Madrid, Spain. He holds a PhD and MSc in Computer Science from ETH Zurich, Switzerland. His personal website is at: https://zistvan.github.io.

REFERENCES

[1] M. J. Amiri, D. Agrawal, and A. E. Abbadi. Caper: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, 2019.

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[3] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn. Corda: an introduction. *R3 CEV, August*, 1:15, 2016.

[4] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[5] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[6] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. Blockchaindb: A shared database on blockchains. *Proceedings of the VLDB Endowment*, 12(11):1597–1609, 2019.

[7] C. Gorenflo, S. Lee, L. Golab, and S. Keshav. FastFabric: Scaling Hyperledger Fabric to 20,000 transactions per second. In *IEEE ICBC*, 2019.

[8] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *Proceedings of the VLDB Endowment*, 13(6).

[9] L. Kuhring, Z. István, A. Sorniotti, and M. Vukolić. Streamchain: Rethinking blockchain for datacenters. *arXiv preprint arXiv:1808.08406*, 2018.

[10] Y. Li, B. Cao, M. Peng, L. Zhang, L. Zhang, D. Feng, and J. Yu. Direct acyclic graph-based ledger for internet of things: Performance and security analysis. *IEEE/ACM Transactions on Networking*, 28(4):1643–1656, 2020.

[11] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, 2008.

[12] S. Popov. The tangle. *White paper*, 1:3, 2018.

[13] T. Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. *Available [online].[Accessed: 4-12-2018]*, 2018.

[14] L. S. Sankar, M. Sindhu, and M. Sethumadhavan. Survey of consensus protocols on blockchain applications. In *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pages 1–5. IEEE, 2017.

[15] F. M. Schuhknecht, A. Sharma, J. Dittrich, and D. Agrawal. Chainifydb: How to blockchainify any data management system. *arXiv preprint arXiv:1912.04820*, 2019.

[16] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*, pages 105–122. ACM, 2019.

Deep Learning Approaches for Text-to-SQL Systems

George Katsogiannis-Meimarakis
katso@athenarc.gr
Athena Research Center
Greece

Georgia Koutrika
georgia@athenarc.gr
Athena Research Center
Greece

ABSTRACT

To bridge the gap between users and data, numerous text-to-SQL systems have been developed that allow users to pose natural language questions over relational databases. Recently, novel text-to-SQL systems are adopting deep learning methods with very promising results. At the same time, several challenges remain open making this area an active and flourishing field of research and development. To make real progress in building text-to-SQL systems, we need to de-mystify what has been done, understand how and when each approach can be used, and, finally, identify the research challenges ahead of us. The purpose of this tutorial is to present recent advances of deep learning techniques for text-to-SQL translation, and to highlight open problems and new research opportunities for researchers and practitioners in the fields of database systems, natural language processing and deep learning.

1 INTRODUCTION

Data is a prevalent part of every business and scientific domain, but its explosive volume and increasing complexity make data querying and exploration challenging even for experts. In an attempt to bridge the gap between users and data, numerous *text-to-SQL systems* have been implemented, both from industry and academia, that enable users to pose *unstructured queries* (using keywords or free-form text) over relational databases [1, 4, 26]. The recent advances on deep neural networks and the creation of two large datasets for training text-to-SQL systems, have led to the emergence of several, novel, text-to-SQL systems that leverage deep learning techniques. These efforts show very promising results. At the same time, several open challenges make this area an active and flourishing field of research and development. It is high time for a systematic study of these solutions.

In this work, we aim at presenting the recent advances in the field of text-to-SQL systems with the adoption of deep learning techniques. We follow a systematic and structured approach. First, we introduce the text-to-SQL problem, explain and categorize its challenges. Then, we present available benchmarks and explain their advantages and shortcomings. We zoom in on the recent advances of deep learning techniques for text-to-SQL translation. We explain the problems they address and their limitations, and we highlight research opportunities on the intersection of database systems, natural language processing and deep learning.

2 THE TEXT-TO-SQL PROBLEM

The text-to-SQL (also known as NL2SQL) problem can be described as follows: *Given a Natural Language Query (NLQ) on a Relational Database (RDB), produce a SQL query equivalent to the NLQ, which is valid for the said RDB.* Several challenges arise, including: ambiguity, schema linking, vocabulary gap and user mistakes.

Ambiguity of natural language queries is one of the most difficult challenges a text-to-SQL system has to cope with. There are several types of ambiguity [3, 27]. For instance, *lexical ambiguity* refers to the case of a single word with multiple meanings (e.g., *"Paris"* can be a city or a person).

On the other hand, *schema linking* is the problem of understanding which parts of the NLQ refer to which parts of the database schema. *Vocabulary gap* refers to the differences between the vocabulary used by the database and the one used by the user. *User mistakes*, such as syntactical or grammatical errors, make the problem even more challenging.

3 TEXT-TO-SQL LANDSCAPE

The problem of translating user queries to SQL has been a holy grail for the database community for over 30 years [4]. In this section, we will give a very brief overview of the earlier approaches, especially those proposed by the database community.

Early database approaches use (a) inverted indexes (like search engines do) to map query keywords to database elements (relations, attributes and values) and (b) the database schema to find how relations in a query should be joined [18]. These approaches use either a schema graph that represents the database relations as nodes and the joins between them as edges [2, 6, 14, 21, 22, 30, 38] or a tuple graph where the nodes are the database tuples [5, 9, 13, 16]. Answers to a query are defined as sub-graphs over the complete graph, comprising a subset of the relations and tuples that contain the query keywords and are connected by the joins between them. NALIR [17] is the first to use a syntactic parse tree to represent a query and map it to the database schema graph. ATHENA [29] employs an ontology to represent a real-world domain (such as finance) and an ontology-to-database mapping, which describes how the ontology elements are mapped to the database objects. DBPal [33] aims at generating join queries based on the information learnt from domain-specific training data, and requires many training examples with different join paths.

4 AVAILABLE BENCHMARKS

Training a deep learning system is a very data-intensive procedure; large amounts of data are required in order to train an accurate model. For this reason, the availability of datasets is the main fuel for the development of deep learning solutions and the text-to-SQL task is no exception. In this section, we will introduce the two major large-scale benchmarks, explain their characteristics as well as highlight their shortcomings.

WikiSQL [39] is *a large crowd-sourced dataset for developing natural language interfaces for relational databases* released along with the Seq2SQL text-to-SQL system. It contains over 25,000 tables gathered from Wikipedia pages and over 80,000 natural language and SQL question pairs, which were created by crowdsourcing. Note that each of WikiSQL's questions is directed to a single table and not to a relational database. This means that the

proposed task is much simpler than the ultimate goal of creating a natural language interface for relational databases. Additionally, the complexity of the queries is very low. There are no JOIN, GROUP BY, UNION, INTERSECTION or other complex SQL elements. We must also note that that WikiSQL contains multiple errors and ambiguities, which might hinder the performance of any model trained on it.

Spider [37] *is a large-scale complex and cross-domain semantic parsing and text-to-SQL dataset* annotated by 11 Yale students. It contains 200 relational databases from 138 different domains along with over 10,000 natural language questions and over 5,000 complex SQL queries. Its queries range from very simple to very hard, using all the common SQL elements, including nested queries. All the above, along with the fact that it was hand-crafted and re-checked are an indicator of its higher quality, compared to WikiSQL, and has led to the development of very promising systems.

5 NL REPRESENTATION

We will now provide an overview of the state-of-the-art techniques for natural language representation in neural networks. The use of neural networks, which can only handle numerical inputs and not raw text, has led to the adoption of word embeddings for numerical word representation. Additionally, in the past few years, the use of language models is blooming, following their rise as an efficient solution for increased performance in NL tasks.

Word embeddings assume that every unique word has a numerical representation that can be different from all other words and at the same time incorporate useful information about the word, and aim at mapping each word to a multidimensional vector. Besides the brute-force creation of one-hot embeddings, researchers have provided highly efficient techniques to create representations that carry the word's meaning and its relationships with other words. Word2Vec [25], GloVe [28] and WordPiece embeddings [34], to name a few, are some famous word embedding techniques that are used in most, if not all, text-to-sql systems.

Language models are a novel and emerging type of pre-trained neural networks for processing NL, that has been shown to excel in NL tasks during the past few years. Note that language models are not a replacement for word embeddings, given that they are neural networks and they still need a way of transforming words to vectors. The way this type of models are created, is that a very large network (10^8 order of magnitude of parameters) is created and is pre-trained on a very large NL dataset (10^9 order of magnitude of words). The pre-trained model is made available for researchers who can then adapt its inputs and outputs to the specific task they aim to solve, and train it for an additional number of epochs on their task-specific dataset. The result is a much stronger model that can reach state-of-the-art performance even without the need of complex architectures [8]. These models have been able to reach such performances due to the use of a neural network architecture that was recently proposed, called the Transformer [31], which excels at handling NL sequences. Some of the most used language models for the text-to-SQL task are BERT [8] and MT-DNN [20].

6 TEXT-TO-SQL DEEP LEARNING APPROACHES

Deep learning systems following the encoder-decoder architecture can be distinguished in three categories, based on the output of their decoder [7]: (a) sequence-to-sequence approaches, (b) grammar-based approaches, and (c) sketch-based slot-filling approaches. In order to better understand the proposed systems, we will now give a taxonomy of deep learning approaches for text-to-SQL, and highlight the main characteristics as well as the advantages and shortcomings of each neural network architecture. Additionally, we will provide an overview of some key systems in each category.

6.1 Sequence-to-sequence approaches

This category includes systems (e.g. [19, 39]) that produce a sequence of SQL tokens and schema elements as their output, with the resulting sequence being the final SQL query prediction, or a major part of it. Essentially, they attempt to transform an input NLQ sequence to an output SQL sequence. This approach is the simplest, but is also very prone to errors. It was adopted by one of the first deep-learning systems for the task at hand, Seq2SQL [39], but later systems steered away from such approaches. The main drawback of sequence-to-sequence architectures is that they do not take the strict grammatical rules of SQL into account when generating a query. The system attempts to learn how a SQL sequence is generated, but at prediction time there are no measures to safeguard from producing syntactically incorrect queries.

Seq2SQL [39] was one of the first neural networks created specifically for the text-to-SQL task and was based on a previous work focusing on generating logical forms using neural networks [10]. Its authors released the WikiSQL dataset along with it, which signified a new era for deep learning research on the text-to-SQL problem. The system predicts an aggregation function and the column for the SELECT clause as classification tasks and generates the WHERE condition clause using a seq-to-seq network. The latter part of the system is burdened with generating parts of the query that can lead to syntactic errors, which is its major drawback. The network architecture combines LSTM and linear layers, and the GloVe embeddings are used to represent the inputs.

6.2 Grammar-based approaches

Grammar-based approaches (e.g., [7, 11, 12, 32]) are an evolution of sequence-to-sequence approaches, and produce a sequence of grammar rules instead of simple tokens as their output. These grammar rules are instructions that, when applied, can create a SQL query. The advantage over sequence-to-sequence approaches is that the possibility for generating an out-of-place token or a syntactically incorrect query is dramatically reduced. This is the most used approach for generating complex SQL queries.

RAT-SQL [32] is a grammar-based text-to-SQL system focusing on the Spider dataset. It is capable of generating complex SQL queries by incorporating three note-worthy features. First, it creates a *question-contextualized schema graph*, i.e. a graph representing the database schema, its tables and columns, as well as the words of the user's question as nodes and the connections between them as edges. The edges between DB elements are created based on the DB schema and the edges between NLQ words and DB elements are created by performing text matching, which is a form of schema linking. Furthermore, it uses a modified Transformer network for *relation aware self-attention*, that is specifically designed to leverage the information of the created graph and its edges. Finally, it follows a method for SQL

generation as an abstract syntax tree, by generating a sequence of actions for building the tree, as proposed in [36].

IRNet [12] is another grammar-based system capable of generating complex SQL queries. It uses text-matching techniques to address the schema linking challenge similarly but in a simpler form than RAT-SQL. It uses a complex architecture of linear and recurrent neural networks to process the input, in addition to BERT. After processing the input, it creates an SQL query using the same method as RAT-SQL for generating an abstract syntax tree, with the main difference that the output it produces is in an intermediate language called SemQL designed specifically for this system. Its authors argue that it is easier to generate queries in this language and then transform them to SQL.

6.3 Sketch-based slot-filling approaches

Systems following this approach (e.g., [15, 23, 24, 35]) aim at simplifying the difficult task of generating a SQL query, to the easier task of predicting certain parts of the query (e.g. which of the table columns will appear in the SELECT clause), transforming in this way the SQL generation task to a classification task. In this case, we consider a query sketch with a number of empty slots that must be filled and develop neural networks that predict which element is most probable to fill each slot. A basic prerequisite for such approaches is to have a query sketch that, when filled, will be able to capture the NLQ's intention. As a result, this category of systems is rarely able to produce complex SQL queries.

SQLNet [35] was one of the first sketch-based approaches. It was based on the observation that the way Seq2SQL chose to generate the WHERE clause was prone to errors that could be avoided. For this reason, a query sketch, which could cover every SQL query in the WikiSQL dataset, was developed and separate neural networks were created to fill each slot. All slots are filled by considering a classification task (e.g., which of the six possible aggregation functions is appropriate for the given NLQ) except for the condition value slot which was generated by a seq-to-seq network. Note that in this case the aforementioned seq-to-seq network only generates a value and does not handle SQL tokens, meaning that it is not possible to generate syntactically incorrect queries. Another improvement is the introduction of a *column attention* neural mechanism to the network.

HydraNet [23] focuses on the WikiSQL task and follows a sketch-based approach, using the same sketch as SQLNet, but takes advantage of the BERT language model and achieves much better results.

SQLova [15] is another sketch-based approach focusing on the WikiSQL dataset and leveraging the BERT language model, just as the HydraNet system. Their main difference is that while HydraNet aims to use a very simple network after receiving BERT's output, SQLova employs a large and complex network similar to the one used by SQLNet, while also incorporating BERT into the system. What must be noted is that even though SQLova employs a larger and more complex network than HydraNet, it achieves lower accuracy scores on the WikiSQL dataset.

7 CHALLENGES AND RESEARCH OPPORTUNITIES

While a lot of progress has been made on the text-to-SQL problem, several important issues need to be tackled. Here, we outline some of the most challenging ones.

The need for new benchmarks and in-depth system evaluations is pressing and the database community can help complement the work done by benchmarks such as Spider. New benchmarks are needed that can test the query expressivity (i.e., what types of queries a system can answer) as well as the efficiency and scalability of text-to-SQL systems to bigger and more complex data sets.

Furthermore, there is a need for further research on answer validation. Since in many cases users are not familiar with SQL, the question is how they can confirm that the obtained results match the intention of the NLQ. Another challenge is the universality of the solution, i.e. the system's ability to perform equally well for different databases. It is also important to enable natural language queries in languages other than English, which is the main focus of current efforts. Due to the problem's multidisciplinarity, database, ML, and NLP approaches can join forces to push the barrier further.

8 PRESENTERS

George Katsogiannis-Meimarakis is a research assistant at Athena Research Center in Athens, Greece, where he works on the INODE (Intelligent Open Data Exploration) project, focusing on the text-to-SQL problem. He is a graduate of the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens, where he completed his thesis with the title "Translating Natural Language to SQL using Deep Learning". Currently, he is attending a MSc programme on Data Science and Information Technologies with a specialisation on Artificial Intelligence and Big Data.

Georgia Koutrika is a Research Director at Athena Research Center in Greece. She has more than 15 years of experience in multiple roles at HP Labs, IBM Almaden, and Stanford. Her work focuses on data exploration, recommendations, and data analytics, and has been incorporated in commercial products, described in 14 granted patents and 26 patent applications in the US and worldwide, and published in more than 90 papers in top-tier conferences and journals. She is Editor-in-chief for VLDB Journal, PC chair for VLDB 2023, associate editor for TKDE, and an ACM Distinguished Speaker. *Prior tutorials:* Fairness in Rankings and Recommenders [EDBT20], Recommender Systems [SIGMOD'18, EDBT'18, ICDE'15], Personalization [ICDE'10, ICDE'07, VLDB'05].

ACKNOWLEDGMENTS

This work has been partially funded by the European Union's Horizon 2020 research and innovation program (grant agreement No 863410).

REFERENCES

[1] Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. 2019. A comparative survey of recent natural language interfaces for databases. *VLDB J.* 28, 5 (2019), 793–819.
[2] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. 2002. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*. 5–16.
[3] Ambiguity [n.d.]. Ambiguity. https://stanford.io/2YXcECi.
[4] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases - an introduction. *Natural Language Engineering* 1, 1 (1995), 29–81. https://doi.org/10.1017/S135132490000005X
[5] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. 2002. Keyword searching and browsing in databases using BANKS. In *ICDE*. 431–440.
[6] Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. 2012. SODA: Generating SQL for Business Users. *PVLDB* 5, 10 (2012), 932–943.

[7] DongHyun Choi, Myeong Cheol Shin, EungGyun Kim, and Dong Ryeol Shin. 2020. RYANSQL: Recursively Applying Sketch-based Slot Fillings for Complex Text-to-SQL in Cross-Domain Databases. arXiv:cs.CL/2004.03125

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:cs.CL/1810.04805

[9] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. 2007. Finding top-k min-cost connected trees in databases. In *ICDE*. 836–845.

[10] Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. arXiv:cs.CL/1601.01280

[11] Li Dong and Mirella Lapata. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. arXiv:cs.CL/1805.04793

[12] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. arXiv:cs.CL/1905.08205

[13] Hao He, Haixun Wang, Jun Yang, and Philip S Yu. 2007. BLINKS: ranked keyword searches on graphs. In *ACM SIGMOD* . ACM, 305–316.

[14] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. 2003. Efficient IR-style Keyword Search over Relational Databases. In *VLDB*. 850–861.

[15] Wonseok Hwang, Jinyeong Yim, Seunghyun Park, and Minjoon Seo. 2019. A Comprehensive Exploration on WikiSQL with Table-Aware Word Contextualization. arXiv:cs.CL/1902.01069

[16] Mehdi Kargar, Aijun An, Nick Cercone, Parke Godfrey, Jaroslaw Szlichta, and Xiaohui Yu. 2014. *MeanKS: Meaningful Keyword Search in Relational Databases with Complex Schema.* ACM. http://ceur-ws.org/Vol-1912/paper20.pdf

[17] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *PVLDB* 8, 1 (Sept. 2014), 73–84.

[18] Yunyao Li and Davood Rafiei. 2017. Natural Language Data Management and Interfaces: Recent Development and Open Challenges. In *ACM SIGMOD*. 1765–1770.

[19] Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging Textual and Tabular Data for Cross-Domain Text-to-SQL Semantic Parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 4870–4888. https://doi.org/10.18653/v1/2020.findings-emnlp.438

[20] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. 2019. Multi-Task Deep Neural Networks for Natural Language Understanding. arXiv:cs.CL/1901.11504

[21] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. 2007. Spark: Top-k Keyword Query in Relational Databases. In *ACM SIGMOD*. 115–126.

[22] Yi Luo, Wei Wang, Xuemin Lin, Xiaofang Zhou, Jianmin Wang, and Keqiu Li. 2011. SPARK2: Top-k Keyword Query in Relational Databases. *IEEE Trans. Knowl. Data Eng.* 23, 12 (2011), 1763–1780. https://doi.org/10.1109/TKDE.2011.60

[23] Qin Lyu, Kaushik Chakrabarti, Shobhit Hathi, Souvik Kundu, Jianwen Zhang, and Zheng Chen. 2020. Hybrid Ranking Network for Text-to-SQL. arXiv:cs.CL/2008.04759

[24] Jianqiang Ma, Zeyu Yan, Shuai Pang, Yang Zhang, and Jianping Shen. 2020. Mention Extraction and Linking for SQL Query Generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 6936–6942. https://doi.org/10.18653/v1/2020.emnlp-main.563

[25] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:cs.CL/1301.3781

[26] Amihai Motro. 1986. Constructing Queries from Tokens. In *ACM SIGMOD*. 120–131. https://doi.org/10.1145/16894.16866

[27] Notes on Ambiguity [n.d.]. Notes on Ambiguity. http://bit.ly/2YTLFeR.

[28] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[29] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, Fatma Özcan, IBM Research. Bangalore, and IBM Research. Almaden. 2016. *ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores.* VLDB. http://www.vldb.org/pvldb/vol9/p1209-saha.pdf

[30] Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. 2008. Précis: from unstructured keywords as queries to structured databases as answers. *The VLDB Journal* 17, 1 (2008), 117–149.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:cs.CL/1706.03762

[32] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. arXiv:cs.CL/1911.04942

[33] Nathaniel Weir, Prasetya Utama, Alex Galakatos, Andrew Crotty, Amir Ilkhechi, Shekar Ramaswamy, Rohin Bhushan, Nadja Geisler, Benjamin Hättasch, Steffen Eger, Ugur Çetintemel, and Carsten Binnig. 2020. DBPal: A Fully Pluggable NL2SQL Training Pipeline. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2347–2361.

[34] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. arXiv:cs.CL/1609.08144

[35] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. arXiv:cs.CL/1711.04436

[36] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. arXiv:cs.CL/1704.01696

[37] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2019. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. arXiv:cs.CL/1809.08887

[38] Zhong Zeng, Mong Li Lee, and Tok Wang Ling. 2016. Answering Keyword Queries involving Aggregates and GROUPBY on Relational Databases. *EDBT* (2016), 161–172.

[39] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. arXiv:cs.CL/1709.00103

Big Sequence Management: Scaling up and Out

Karima Echihabi
Mohammed VI Polytechnic University
karima.echihabi@um6p.ma

Kostas Zoumpatianos*
LIPADE, Université de Paris
konstantinos.zoumpatianos@u-paris.fr

Themis Palpanas
LIPADE, Université de Paris
French University Institute (IUF)
themis@mi.parisdescartes.fr

ABSTRACT

Data series are a prevalent data type that has attracted lots of interest in recent years. Specifically, there has been an explosive interest towards the analysis of large volumes of data series in many different domains. This is both in businesses (e.g., in mobile applications) and in sciences (e.g., in biology). In this tutorial, we focus on applications that produce massive collections of data series, and we provide the necessary background on data series storage, retrieval and analytics. We look at systems historically used to handle and mine data in the form of data series, as well as at the state of the art data series management systems that were recently proposed. Moreover, we discuss the need for fast similarity search for supporting data mining applications, and describe efficient similarity search techniques, indexes and query processing algorithms. Finally, we look at the gap of modern data series management systems in regards to support for efficient complex analytics, and we argue in favor of the integration of summarizations and indexes in modern data series management systems. We conclude with the challenges and open research problems in this domain.

1 INTRODUCTION

In various scientific and industrial domains analysts are required to measure quantities as they fluctuate over a dimension; these values are commonly called *data series* or *sequences*. The dimension over which data series are ordered depends on the application domain and can have various diverse physical meanings. By far, the most common dimension over which data are ordered is time. In this case, we specifically talk about *time series*. Other applications though, produce series ordered over position (DNA sequences), mass (mass spectrometry) or angle (shapes). In all cases, data have to be captured, stored and analyzed as series rather than individual values.

Applications range from forecasting methods to correlation analysis, summarization, representation methods, sampling, outlier detection and more [6–8, 35, 41]. Moreover, it is not unusual for applications to involve numbers of sequences in the order of hundreds of millions to billions [1, 3]. As a result, analysts are more frequently than ever deluged by the vast amounts of data series that they have to filter, process and understand. Consider for instance, that for several of their analysis tasks, neuroscientists are currently reducing each of their 3,000 point long sequences to a single number (the global average) in order to be able to analyze their huge datasets [1]. In astronomy, there are currently available more than 70TB of spectroscopic sequence data from 200 million sky objects, collected by the Sloan Digital Sky Survey [3], allowing scientists to study the universe. These data have to be processed and analyzed, in order to identify patterns, gain insights, and detect abnormalities.

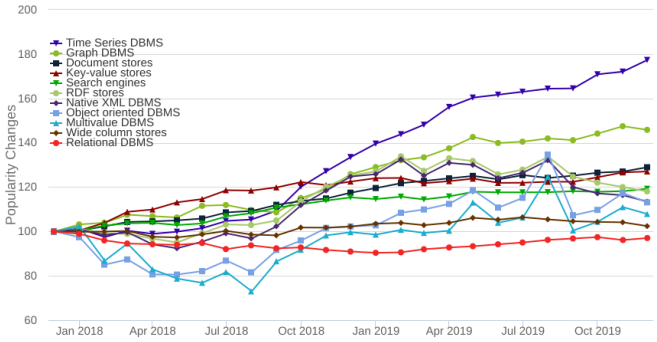


Figure 1: DBMS category popularity change trend [2]

Recent advances in domains such as cloud computing and data centers, IoT and smart cities, self-driving cars and communications, generated a tremendous interest in developing specialized systems able to manage and mine data series. This is evident both by industrial [42] [17] and academic interest [5, 28], as well as through popularity studies [2], where time series management systems gather the most intense interest change over the last two years, as shown in Figure 1.

Our goal is to describe the current state in data series management, including applications, query types and data types, complex analytic algorithms, their components and their implementation in modern data systems. Further on we will explore how modern techniques can be leveraged to speed up complex analytical pipelines, and take a glimpse on how these techniques can be improved by applying machine learning.

2 SEQUENCE MANAGEMENT OVERVIEW

We take a holistic look at the problem of managing and analyzing very large collections of data series, discuss the state-of-the-art and pinpoint the opportunities for optimizing complex query execution.

[Introduction and Foundations] We will start by looking at some foundational aspects of data series management. Those include the *data characteristics*, the query workloads, and the *specialized data structures* used to index sequential data. Data series can be categorized under many dimensions: i.e., the way that data arrive: streaming vs static, the lengths of data series: fixed vs variable length per series, the way that points are sampled: fixed intervals vs variable sampling intervals, and the presence of uncertainty in their values.

In terms of workloads, we will then look at various applications and query patterns that recur in each one of those. Specifically, we will discussing both simple Selection-Projection-Transformation (SPT) queries, where analysts filter based on data properties (e.g.,

*The author is currently at Snowflake Computing

thresholds) or meta-data values, as well as complex data mining (DM) analytics, like clustering, outlier detection and more [39]. We will look at the core component of advanced analytics, which is similarity search, and look at the different flavors of this problem. Those include whole matching vs sub-sequence matching, exact vs approximate similarity search, as well as various distance measures that are commonly used in practice. Finally, we will briefly talk about the different data structure categories that exist, and how they are used to organize and retrieve data in each one of the aforementioned query patterns.

[Complex Analytics] We will dive in analytics like outlier detection [12, 16], frequent pattern mining [51], clustering [29, 52, 54, 63], and classification [13]. Such analytics involve a series of operations that are performed in a pre-processing step (e.g., sliding windows, normalization, interpolation, etc.), as well as operations that are repeated in the context of an iterative algorithm (e.g., similarity search). We will discuss these operations, and pinpoint the ones that can be optimized at the database kernel level. Such operations include sliding windows, normalization, interpolation, and various transformations such as dft that are specific to each algorithm. During the iterative part of these analytics, multiple similarity search operations need to be performed. This is useful for finding series within a given radius from a centroid in clustering, or for identifying distances from a given model in anomaly detection and classification, but also for retrieving patterns in frequent pattern mining. All of these operations can be implemented externally, in the application side. However, since some of them are data-intensive, pruning or incremental computation can significantly improve their performance. For this reason, performing them at the database level can provide large improvements in terms of execution time. We will focus on similarity search as such an example, being a crucial and expensive component of most mining algorithms, and motivate a deep-dive at its characteristics and scalable implementations.

[Systems for Data Series Management] We will then look at current state-of-the-art systems, describing their storage layers and data structures, as well as how they implement the aforementioned data manipulation operations. In particular, we will both look at systems that have been specifically designed to support sequential data, as well as systems that have been adapted to support them.

Specialized systems either utilize custom storage layers, or existing solutions. Common off-the-shelf storage systems are log-structure merge tree (LSM) based engines like RocksDB and LevelDB, and distributed systems such as HBase. Custom engines utilize domain-specific compression, indexing and data partitioning to increase efficiency. They support both simple and complex analytical queries and some of the systems offer encryption and distributed query processing.

Beringei [42] is developed by Facebook, it has a custom in-memory storage engine. It compresses and organizes data in a series per series scheme. CrateDB [15] partitions data in chunks, stores them in a distributed file system, and indexes them using Apache Lucene. InfluxDB [27] uses Time-Structured Merge Trees (LSM tree variant), logging data on disk as they arrive, and periodically merge-sorting overlapping time-stamps. Prometheus [48] is based on the Beringei ideas. QuasarDB [49] utilizes either RocksDB or Hellium [26]. Riak TS [53] supports both LevelDB or Bitcask, which is a custom log structured hash table. Timescale [59] is a Postgres extension. It partitions time series both in groups of series as well as in distinct time segments. It then provides an abstraction of a single table. Finally, various systems such as OpenTSDB [38], Timely [58] (concentrated on security) and Warp10 [62] are developed on top of HBase.

All the aforementioned systems support range scans in the positions, aggregation functions and filtering. Beringei additionally supports correlation queries through a brute force implementation. Crate supports geospatial queries. InfluxDB supports queries like moving averages, prediction, transformations, etc, and Timescale supports gap filling.

[Advanced Techniques for Optimizing Analytics] We will present techniques for speeding up similarity search, which plays a central role in several algorithms related to complex data series analytics, and discuss opportunities for integrating such techniques in modern data series management systems. Previous work on similarity search has proposed the use of spatial indexes such as R-Trees with DFT [4, 50] and DHWT [11]. Specialized indexes are based on domain specific summarizations. Examples include DS-Tree [61], *i*SAX [40, 56], *i*SAX 2.0 [9], *i*SAX2+ [10], ADS+ [68, 68], SFA [55], Coconut [31, 32], and ULISSE [33, 34].

In addition, we will pay particular attention to parallel and distributed solutions for similarity search. These include methods that support both exact and approximate similarity search query answering, and make use of modern hardware (e.g., SIMD, multi-core, multi-socket, GPU) such as ParIS+ [43, 45], Delta-Top-Index [47], MESSI [44], and SING [46], as well as distributed computation (e.g., Spark) such as DPiSAX [65, 66], TARDIS [67], KV-match [64], MVS-match [23], and L-match [22]. These methods are in a much better position than traditional single-node techniques to address the scalability challenges of modern data series analytics applications that have to deal with very large data collections.

Apart from exact indexes, there are also various approximate index structures proposed in the literature. Those include methods based on hashing [30, 57], sketches and grid indexes [14], and kNN-Graphs [36, 37]. Recent studies [20, 21] have compared several data series and high-dimensional similarity search methods under a common framework, revealing multiple promising future research directions, which we will analyze.

[Challenges and Conclusions] Massive data series collections are becoming a reality for virtually every scientific and social domain. This leads to the need of designing and developing general-purpose Data Series Management Systems, able to cope with big data series, that is, very large and fast-changing collections of data series, which can be heterogeneous (i.e., originate from disparate domains and thus exhibit very different characteristics), and which can have uncertainty in their values (e.g., due to inherent errors in the measurements). These systems should have data series indexes and summarizations integrated into their engines, so as to speedup the time-intensive operations of complex analytics pipelines, and support interactive exploration of big data series. To this end, progressive analytics operators would also be very useful [24, 25, 60]. At the same time, the role that deep learning techniques can play should be studied in more detail, especially with regards to similarity search [18, 19] and query optimization. Finally, there is a pressing need for developing data series specific benchmarks [69, 70] able to stress test index structures in a principled way.

REFERENCES

[1] [n.d.]. ADHD-200. http://fcon_1000.projects.nitrc.org/indi/adhd200/.
[2] [n.d.]. DB-Engines. https://db-engines.com/en/ranking_categories.
[3] [n.d.]. Sloan Digital Sky Survey. https://www.sdss3.org/dr10/data_access/\rvolume.php.

[4] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. 1993. Efficient Similarity Search In Sequence Databases. In *FODO*.

[5] Andreas Bader, Oliver Kopp, and Michael Falkenthal. 2017. Survey and Comparison of Open Source Time Series Databases. In *BTW*.

[6] Anthony J. Bagnall, Richard L. Cole, Themis Palpanas, and Konstantinos Zoumpatianos. 9(7), 2019. Data Series Management (Dagstuhl Seminar 19282). *Dagstuhl Reports* (9(7), 2019).

[7] Paul Boniol, Michele Linardi, Federico Roncallo, and Themis Palpanas. 2020. Automated Anomaly Detection in Large Sequences. In *ICDE*.

[8] Paul Boniol and Themis Palpanas. 2020. Series2Graph: Graph-based Subsequence Anomaly Detection for Time Series. *PVLDB* (2020).

[9] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn J. Keogh. 2010. iSAX 2.0: Indexing and Mining One Billion Time Series. In *ICDM*.

[10] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn J. Keogh. 2014. Beyond one billion time series: indexing and mining very large time series collections with iSAX2+. *KAIS* 39, 1 (2014), 123–151.

[11] Kin-pong Chan and Ada Wai-Chee Fu. 1999. Efficient Time Series Matching by Wavelets. In *ICDE*.

[12] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)* 41, 3 (2009), 15. http://scholar.google.de/scholar.bib?q=info:jAfBmk-9uAcJ:scholar.google.com/&output=citation&hl=de&ct=citation&cd=0

[13] Yihua Chen, Eric K. Garcia, Maya R. Gupta, Ali Rahimi, and Luca Cazzanti. 2009. Similarity-based Classification: Concepts and Algorithms. *J. Mach. Learn. Res.* 10 (June 2009), 747–776. http://dl.acm.org/citation.cfm?id=1577069.1577096

[14] Richard Cole, Dennis E. Shasha, and Xiaojian Zhao. 2005. Fast window correlations over uncooperative time series. In *KDD*.

[15] Crate. 2018. CrateDB: Real-time SQL Database for Machine Data & IoT. http://crate.io/

[16] Michele Dallachiesa, Themis Palpanas, and Ihab F. Ilyas. 2014. Top-k Nearest Neighbor Search in Uncertain Data Series. *PVLDB* 8, 1 (2014).

[17] Lars Dannecker, Gordon Gaumnitz, Boyi Ni, and Yu Cheng. 2015. Multi-representation Storage of Time Series Data. US Patent 20170161340A1.

[18] Karima Echihabi. 2020. High-Dimensional Vector Similarity Search: From Time Series to Deep Network Embeddings. In *SIGMOD*.

[19] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2020. Scalable Machine Learning on High-Dimensional Vectors: From Data Series to Deep Network Embeddings. In *WIMS*.

[20] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2018. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB* 12, 2 (2018).

[21] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2019. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *PVLDB* (2019).

[22] Kefeng Feng, Peng Wang, Jiaye Wu, and Wei Wang. 2020. L-Match: A Lightweight and Effective Subsequence Matching Approach. *IEEE Access* 8 (2020), 71572–71583.

[23] Kefeng Feng, Jiaye Wu, Peng Wang, Ningting Pan, and Wei Wang. 2019. MVS-match: An Efficient Subsequence Matching Approach Based on the Series Synopsis. In *DASFAA*, Vol. 11448. Springer, 368–372.

[24] Anna Gogolou, Theophanis Tsandilas, Karima Echihabi, Themis Palpanas, and Anastasia Bezerianos. 2020. Data Series Progressive Similarity Search with Probabilistic Quality Guarantees. In *SIGMOD*.

[25] Anna Gogolou, Theophanis Tsandilas, Themis Palpanas, and Anastasia Bezerianos. 2019. Progressive Similarity Search on Time Series Data. In *Proceedings of the Workshops of the EDBT/ICDT 2019 Joint Conference, EDBT/ICDT*.

[26] Hellium. 2018. Hellium: Ultra high performance key/value storage. https://www.levyx.com/helium

[27] InfluxDB. 2018. InfluxDB - Open Source Time Series, Metrics, and Analytics Database (http://influxdb.com/). http://influxdb.com/

[28] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time Series Management Systems: A Survey. *TKDE* 29, 11 (2017).

[29] Eamonn Keogh and M. Pazzani. 1998. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In *KDD*, R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro (Eds.). ACM Press, New York City, NY, 239–241.

[30] Yongwook Bryce Kim. 2017. *Physiological time series retrieval and prediction with locality-sensitive hashing*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, USA.

[31] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. *PVLDB* 11, 6 (2018), 677–690. https://doi.org/10.14778/3184470.3184472

[32] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2019. Coconut: sortable summarizations for scalable indexes over static and streaming data series. *VLDBJ* 28, 6 (2019).

[33] Michele Linardi and Themis Palpanas. 2018. Scalable, Variable-Length Similarity Search in Data Series: The ULISSE Approach. *PVLDB* 11, 13 (2018), 2236–2248.

[34] Michele Linardi and Themis Palpanas. 2018. ULISSE: ULtra compact Index for Variable-Length Similarity SEarch in Data Series. In *ICDE*.

[35] Michele Linardi, Yan Zhu, Themis Palpanas, and Eamonn J. Keogh. 2020. Matrix Profile Goes MAD: Variable-Length Motif And Discord Discovery in Data Series. *DAMI*.

[36] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.* 45 (2014), 61–68.

[37] Yury A. Malkov and D. A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *CoRR* abs/1603.09320 (2016).

[38] OpenTSDB. 2015. OpenTSDB - A Distributed, Scalable Monitoring System (http://opentsdb.net/). http://opentsdb.net/

[39] Themis Palpanas. 2015. Data Series Management: The Road to Big Sequence Analytics. *SIGMOD Rec.* 44, 2 (2015), 47–52.

[40] Themis Palpanas. 2020. Evolution of a Data Series Index. *CCIS* 1197 (2020).

[41] Themis Palpanas and Volker Beckmann. 48(3), 2019. Report on the First and Second Interdisciplinary Time Series Analysis Workshop (ITISA). *SIGREC* (48(3), 2019).

[42] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *PVLDB* 8, 12 (2015), 1816–1827.

[43] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2018. ParIS: The Next Destination for Fast Data Series Indexing and Query Answering.

[44] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2020. MESSI: In-Memory Data Series Indexing. In *ICDE*.

[45] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2020. ParIS+: Data Series Indexing on Multi-core Architectures. *TKDE* (2020).

[46] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2021. SING: Sequence Indexing Using GPUs. In *ICDE*.

[47] Danila Piatov, Sven Helmer, Anton Dignös, and Johann Gamper. 2019. Interactive and space-efficient multi-dimensional time series subsequence matching. *Inf. Syst.* 82 (2019), 121–135.

[48] Prometheus. 2018. Prometheus – Monitoring system & time series database. http://prometheus.io/

[49] QuasarDB. 2018. QuasarDB: high-performance, distributed, time series database. https://www.quasardb.net/

[50] Davood Rafiei and Alberto O. Mendelzon. 1997. Similarity-Based Queries for Time Series Data. In *SIGMOD*.

[51] Thanawin Rakthanmanon, Bilson J. L. Campana, Abdullah Mueen, Gustavo Batista, M. Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn J. Keogh. 2012. Searching and mining trillions of time series subsequences under dynamic time warping.. In *KDD*.

[52] Thanawin Rakthanmanon, Eamonn J Keogh, Stefano Lonardi, and Scott Evans. 2011. Time series epenthesis: Clustering time series streams requires ignoring some data. In *ICDM*.

[53] RiakTS. 2018. Riak TS – Basho Technologies. http://basho.com/products/riak-ts/

[54] Pedro Pereira Rodrigues, João Gama, and João Pedro Pedroso. 2006. ODAC: Hierarchical Clustering of Time Series Data Streams.. In *SDM*, Joydeep Ghosh, Diane Lambert, David B. Skillicorn, and Jaideep Srivastava (Eds.). SIAM, 499–503. http://dblp.uni-trier.de/db/conf/sdm/sdm2006.html#RodriguesGP06

[55] Patrick Schäfer and Mikael Högqvist. 2012. SFA: a symbolic fourier approximation and index for similarity search in high dimensional datasets. In *EDBT*.

[56] Jin Shieh and Eamonn J. Keogh. 2008. *i*SAX: indexing and mining terabyte sized time series. In *KDD*.

[57] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: Solving c-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *PVLDB* 8, 1 (2014), 1–12.

[58] Timely. 2018. Timely – A secure time series database based on Accumulo and Grafana. https://code.nsa.gov/timely/

[59] Timescale. 2018. Timescale - an open source time series management system. http://timescale.com/

[60] Cagatay Turkay, Nicola Pezzotti, Carsten Binnig, Hendrik Strobelt, Barbara Hammer, Daniel A. Keim, Jean-Daniel Fekete, Themis Palpanas, Yunhai Wang, and Florin Rusu. 2018. Progressive Data Science: Potential and Challenges. *CoRR* abs/1812.08032 (2018). arXiv:1812.08032 http://arxiv.org/abs/1812.08032

[61] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A Data-adaptive and Dynamic Segmentation Index for Whole Matching on Time Series. *PVLDB* 6, 10 (2013), 793–804.

[62] Warp10. 2018. Warp 10 – The Most Advanced Time Series Platform. https://www.warp10.io/

[63] T. Warren Liao. 2005. Clustering of time series data—a survey. *Pattern Recognition* 38, 11 (2005), 1857–1874.

[64] Jiaye Wu, Peng Wang, Ningting Pan, Chen Wang, Wei Wang, and Jianmin Wang. 2019. KV-Match: A Subsequence Matching Approach Supporting Normalization and Time Warping. In *ICDE*.

[65] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. 2017. DPiSAX: Massively Distributed Partitioned iSAX. In *ICDM*. 1135–1140.

[66] Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2020. Massively Distributed Time Series Indexing and Querying. *TKDE* 32, 1 (2020).

[67] Liang Zhang, Noura Alghamdi, Mohamed Y Eltabakh, and Elke A Rundensteiner. 2019. TARDIS: Distributed indexing framework for big time series data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1202–1213.

[68] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2016. ADS: the adaptive data series index. *VLDB J.* 25, 6 (2016), 843–866.

[69] Kostas Zoumpatianos, Yin Lou, Ioana Ileana, Themis Palpanas, and Johannes Gehrke. 2018. Generating data series query workloads. *VLDB J.* 27, 6 (2018).
[70] Kostas Zoumpatianos, Yin Lou, Themis Palpanas, and Johannes Gehrke. 2015. Query Workloads for Data Series Indexes. In *KDD*.

Karima Echihabi is an Assistant Professor at Mohammed VI Polytechnic University (UM6P) in Morocco. She is interested in scalable data analytics and data series management and has performed an extensive analysis of data series indexes. She holds a PhD degree from Mohammed V University (Morocco) and the University of Paris (France) and a Masters Degree in Computer Science from the University of Toronto. She has worked as a software engineer in the Windows team at Microsoft, Redmond (USA), and the Query Optimizer team at the IBM Toronto Lab (Canada).

Kostas Zoumpatianos is a Software Engineer at Snowflake Computing. He has been a Marie Curie Fellow at the University of Paris and a postdoctoral researcher at Harvard University. He got his PhD from the University of Trento in topics related to indexing and managing large collections of data series. He also holds a M.Sc. in Information Management and a Dipl.Eng. in Information and Communication Systems Engineering from the University of the Aegean in Greece.

Themis Palpanas is Senior Member of the French University Institute (IUF), a distinction that recognizes excellence across all academic disciplines, and professor of computer science at the University of Paris (France), where he is director of the Data Intelligence Institute of Paris (diiP), and director of the data management group, diNo. He received the BS degree from the National Technical University of Athens, Greece, and the MSc and PhD degrees from the University of Toronto, Canada. His interests include problems related to data science (big data analytics and machine learning applications). He is the author of 9 US patents and 2 French patents. He is the recipient of 3 Best Paper awards, and the IBM Shared University Research (SUR) Award. He is currently serving on the VLDB Endowment Board of Trustees, and as an Editor in Chief for the BDR Journal. He has served as General Chair for VLDB 2013, and in the program committees of all major conferences in the areas of data management and analysis.